

## Source Code Listing (Volume 1)

```

/*
 * jcapimin.c
 *
 * Copyright (C) 1994-1998, Thomas G. Lane.
 * This file is part of the Independent JPEG Group's software.
 * For conditions of distribution and use, see the accompanying README file.
 *
 * This file contains application interface code for the compression half
 * of the JPEG library. These are the "minimum" API routines that may be
 * needed in either the normal full-compression case or the transcoding-only
 * case.
 *
 * Most of the routines intended to be called directly by an application
 * are in this file or in jcapistd.c. But also see jcparam.c for
 * parameter-setup helper routines, jcomapi.c for routines shared by
 * compression and decompression, and jctrans.c for the transcoding case.
 */

#define JPEG_INTERNALS
#include "jinclude.h"
#include "jpeglib.h"

/*
 * Initialization of a JPEG compression object.
 * The error manager must already be set up (in case memory manager fails).
 */

GLOBAL(void)
jpeg_CreateCompress (j_compress_ptr cinfo, int version, size_t structsize)
{
  int i;

  /* Guard against version mismatches between library and caller. */
  cinfo->mem = NULL; /* so jpeg_destroy knows mem mgr not called */
  if (version != JPEG_LIB_VERSION)
    ERREXIT2(cinfo, JERR_BAD_LIB_VERSION, JPEG_LIB_VERSION, version);
  if (structsize != SIZEOF(struct jpeg_compress_struct))
    ERREXIT2(cinfo, JERR_BAD_STRUCT_SIZE,
      (int) SIZEOF(struct jpeg_compress_struct), (int) structsize);

  /* For debugging purposes, we zero the whole master structure.
   * But the application has already set the err pointer, and may have set
   * client_data, so we have to save and restore those fields.
   * Note: if application hasn't set client_data, tools like Purify may
   * complain here.
   */
  struct jpeg_error_mgr * err = cinfo->err;
  void * client_data = cinfo->client_data; /* ignore Purify complaint here */
  MEMZERO(cinfo, SIZEOF(struct jpeg_compress_struct));
  cinfo->err = err;
  cinfo->client_data = client_data;

  cinfo->is_decompressor = FALSE;

  /* Initialize a memory manager instance for this object */
  jinit_memory_mgr((j_common_ptr) cinfo);

  /* Zero out pointers to permanent structures. */
  cinfo->progress = NULL;
  cinfo->dest = NULL;

  cinfo->comp_info = NULL;

  for (i = 0; i < NUM_QUANT_TBLS; i++)
    cinfo->quant_tbl_ptrs[i] = NULL;

  for (i = 0; i < NUM_HUFF_TBLS; i++) {
    cinfo->dc_huff_tbl_ptrs[i] = NULL;
    cinfo->ac_huff_tbl_ptrs[i] = NULL;
  }

  cinfo->script_space = NULL;

  cinfo->input_gamma = 1.0; /* in case application forgets */

  /* OK, I'm ready */
  cinfo->global_state = CSTATE_START;
}

```

```

/*
 * Destruction of a JPEG compression object
 */

GLOBAL(void)
jpeg_destroy_compress (j_compress_ptr cinfo)
{
    jpeg_destroy((j_common_ptr) cinfo); /* use common routine */
}

/*
 * Abort processing of a JPEG compression operation,
 * but don't destroy the object itself.
 */

GLOBAL(void)
jpeg_abort_compress (j_compress_ptr cinfo)
{
    jpeg_abort((j_common_ptr) cinfo); /* use common routine */
}

/*
 * Forcibly suppress or un-suppress all quantization and Huffman tables.
 * Marks all currently defined tables as already written (if suppress)
 * or not written (if !suppress). This will control whether they get emitted
 * by a subsequent jpeg_start_compress call.
 *
 * This routine is exported for use by applications that want to produce
 * abbreviated JPEG datastreams. It logically belongs in jparam.c, but
 * since it is called by jpeg_start_compress, we put it here --- otherwise
 * jparam.o would be linked whether the application used it or not.
 */

GLOBAL(void)
jpeg_suppress_tables (j_compress_ptr cinfo, boolean suppress)
{
    int i;
    QUANT_TBL * qtbl;
    HUFF_TBL * htbl;

    for (i = 0; i < NUM_QUANT_TBLS; i++) {
        if ((qtbl = cinfo->quant_tbl_ptrs[i]) != NULL)
            qtbl->sent_table = suppress;
    }

    for (i = 0; i < NUM_HUFF_TBLS; i++) {
        if ((htbl = cinfo->dc_huff_tbl_ptrs[i]) != NULL)
            htbl->sent_table = suppress;
        if ((htbl = cinfo->ac_huff_tbl_ptrs[i]) != NULL)
            htbl->sent_table = suppress;
    }
}

/*
 * Finish JPEG compression.
 *
 * If a multipass operating mode was selected, this may do a great deal of
 * work including most of the actual output.
 */

GLOBAL(void)
jpeg_finish_compress (j_compress_ptr cinfo)
{
    JDIMENSION iMCU_row;

    if (cinfo->global_state == CSTATE_SCANNING ||
        cinfo->global_state == CSTATE_RAW_OK) {
        /* Terminate first pass */
        if (cinfo->next_scanline < cinfo->image_height)
            ERREXIT(cinfo, JERR_TOO_LITTLE_DATA);
        (*cinfo->master->finish_pass) (cinfo);
    } else if (cinfo->global_state != CSTATE_WRCOEFS)
        ERREXIT1(cinfo, JERR_BAD_STATE, cinfo->global_state);
    /* Perform any remaining passes */
    while (! cinfo->master->is_last_pass) {

```

```

(*cinfo->master->prepare_pass) (cinfo);
for (iMCU_row = 0; iMCU_row < cinfo->total_iMCU_rows; iMCU_row++) {
    if (cinfo->progress != NULL) {
        cinfo->progress->pass_counter = (long) iMCU_row;
        cinfo->progress->pass_limit = (long) cinfo->total_iMCU_rows;
        (*cinfo->progress->progress_monitor) ((j_common_ptr) cinfo);
    }
    /* We bypass the main controller and invoke coef controller directly;
     * all work is being done from the coefficient buffer.
     */
    if (! (*cinfo->coef->compress_data) (cinfo, (JSAMPIMAGE) NULL))
        ERREXIT(cinfo, JERR_CANT_SUSPEND);
    (*cinfo->master->finish_pass) (cinfo);
}
/* Write EOI, do final cleanup */
(*cinfo->marker->write_file_trailer) (cinfo);
(*cinfo->dest->term_destination) (cinfo);
/* We can use jpeg_abort to release memory and reset global_state */
jpeg_abort((j_common_ptr) cinfo);
}

/*
 * Write a special marker.
 * This is only recommended for writing COM or APPn markers.
 * Must be called after jpeg_start_compress() and before
 * first call to jpeg_write_scanlines() or jpeg_write_raw_data().
 */

GLOBAL(void)
jpeg_write_marker (j_compress_ptr cinfo, int marker,
                  const JOCTET *dataptr, unsigned int datalen)
{
    METHOD(void, write_marker_byte, (j_compress_ptr info, int val));

    if (cinfo->next_scanline != 0 ||
        (cinfo->global_state != CSTATE_SCANNING &&
         cinfo->global_state != CSTATE_RAW_OK &&
         cinfo->global_state != CSTATE_WRCOEFS))
        ERREXIT1(cinfo, JERR_BAD_STATE, cinfo->global_state);

    (*cinfo->marker->write_marker_header) (cinfo, marker, datalen);
    write_marker_byte = cinfo->marker->write_marker_byte; /* copy for speed */
    while (datalen-- > 0) {
        (*write_marker_byte) (cinfo, *dataptr);
        dataptr++;
    }

    /* Same, but piecemeal. */

GLOBAL(void)
jpeg_write_m_header (j_compress_ptr cinfo, int marker, unsigned int datalen)
{
    if (cinfo->next_scanline != 0 ||
        (cinfo->global_state != CSTATE_SCANNING &&
         cinfo->global_state != CSTATE_RAW_OK &&
         cinfo->global_state != CSTATE_WRCOEFS))
        ERREXIT1(cinfo, JERR_BAD_STATE, cinfo->global_state);

    (*cinfo->marker->write_marker_header) (cinfo, marker, datalen);
}

GLOBAL(void)
jpeg_write_m_byte (j_compress_ptr cinfo, int val)
{
    (*cinfo->marker->write_marker_byte) (cinfo, val);
}

/*
 * Alternate compression function: just write an abbreviated table file.
 * Before calling this, all parameters and a data destination must be set up.
 *
 * To produce a pair of files containing abbreviated tables and abbreviated
 * image data, one would proceed as follows:
 *
 *     initialize JPEG object
 *     set JPEG parameters

```



```

*      set destination to table file
*      jpeg_write_tables(cinfo)
*      set destination to image file
*      jpeg_start_compress(cinfo, FALSE);
*      write data...
*      jpeg_finish_compress(cinfo);
*
* jpeg_write_tables has the side effect of marking all tables written
* (same as jpeg_suppress_tables(..., TRUE)). Thus a subsequent start_compress
* will not re-emit the tables unless it is passed write_all_tables=TRUE.
*/

```

```

GLOBAL(void)
jpeg_write_tables (j_compress_ptr cinfo)
{
    if (cinfo->global_state != CSTATE_START)
        ERREXIT1(cinfo, JERR_BAD_STATE, cinfo->global_state);

    /* (Re)initialize error mgr and destination modules */
    (*cinfo->err->reset_error_mgr) ((j_common_ptr) cinfo);
    (*cinfo->dest->init_destination) (cinfo);
    /* Initialize the marker writer ... bit of a crock to do it here. */
    jinit_marker_writer(cinfo);
    /* Write them tables! */
    (*cinfo->marker->write_tables_only) (cinfo);
    /* And clean up. */
    (*cinfo->dest->term_destination) (cinfo);
    /*
     * In library releases up through v6a, we called jpeg_abort() here to free
     * any working memory allocated by the destination manager and marker
     * writer. Some applications had a problem with that: they allocated space
     * of their own from the library memory manager, and didn't want it to go
     * away during write_tables. So now we do nothing. This will cause a
     * memory leak if an app calls write_tables repeatedly without doing a full
     * compression cycle or otherwise resetting the JPEG object. However, that
     * seems less bad than unexpectedly freeing memory in the normal case.
     * An app that prefers the old behavior can call jpeg_abort for itself after
     * each call to jpeg_write_tables().
     */
}

```

```

/*
 * jcapistd.c
 *
 * Copyright (C) 1994-1996, Thomas G. Lane.
 * This file is part of the Independent JPEG Group's software.
 * For conditions of distribution and use, see the accompanying README file.
 *
 * This file contains application interface code for the compression half
 * of the JPEG library. These are the "standard" API routines that are
 * used in the normal full-compression case. They are not used by a
 * transcoding-only application. Note that if an application links in
 * jpeg_start_compress, it will end up linking in the entire compressor.
 * We thus must separate this file from jcapimin.c to avoid linking the
 * whole compression library into a transcoder.
 */

#define JPEG_INTERNALS
#include "jinclude.h"
#include "jpeglib.h"

/*
 * Compression initialization.
 * Before calling this, all parameters and a data destination must be set up.
 *
 * We require a write_all_tables parameter as a failsafe check when writing
 * multiple datastreams from the same compression object. Since prior runs
 * will have left all the tables marked sent_table=TRUE, a subsequent run
 * would emit an abbreviated stream (no tables) by default. This may be what
 * is wanted, but for safety's sake it should not be the default behavior:
 * programmers should have to make a deliberate choice to emit abbreviated
 * images. Therefore the documentation and examples should encourage people
 * to pass write_all_tables=TRUE; then it will take active thought to do the
 * wrong thing.
 */

GLOBAL(void)
jpeg_start_compress (j_compress_ptr cinfo, boolean write_all_tables)
{
    if (cinfo->global_state != CSTATE_START)
        ERREXIT1(cinfo, JERR_BAD_STATE, cinfo->global_state);
    if (write_all_tables)
        jpeg_suppress_tables(cinfo, FALSE); /* mark all tables to be written */
    /* (Re)initialize error mgr and destination modules */
    (*cinfo->err->reset_error_mgr) ((j_common_ptr) cinfo);
    (*cinfo->dest->init_destination) (cinfo);
    /* Perform master selection of active modules */
    jinit_compress_master(cinfo);
    /* Set up for the first pass */
    (*cinfo->master->prepare_for_pass) (cinfo);
    /* Ready for application to drive first pass through jpeg_write_scanlines
     * or jpeg_write_raw_data.
     */
    cinfo->next_scanline = 0;
    cinfo->global_state = (cinfo->raw_data_in ? CSTATE_RAW_OK : CSTATE_SCANNING);
}

/*
 * Write some scanlines of data to the JPEG compressor.
 *
 * The return value will be the number of lines actually written.
 * This should be less than the supplied num_lines only in case that
 * the data destination module has requested suspension of the compressor,
 * or if more than image_height scanlines are passed in.
 *
 * Note: we warn about excess calls to jpeg_write_scanlines() since
 * this likely signals an application programmer error. However,
 * excess scanlines passed in the last valid call are *silently* ignored,
 * so that the application need not adjust num_lines for end-of-image
 * when using a multiple-scanline buffer.
 */

GLOBAL(JDIMENSION)
jpeg_write_scanlines (j_compress_ptr cinfo, JSAMPARRAY scanlines,
                     JDIMENSION num_lines)
{
    JDIMENSION row_ctr, rows_left;

```

```

if (cinfo->global_state != CSTATE_SCANNING)
    ERREXIT1(cinfo, JERR_BAD_STATE, cinfo->global_state);
if (cinfo->next_scanline >= cinfo->image_height)
    WARNMS(cinfo, JWRN_TOO_MUCH_DATA);

/* Call progress monitor hook if present */
if (cinfo->progress != NULL) {
    cinfo->progress->pass_counter = (long) cinfo->next_scanline;
    cinfo->progress->pass_limit = (long) cinfo->image_height;
    (*cinfo->progress->progress_monitor) ((j_common_ptr) cinfo);
}

/* Give master control module another chance if this is first call to
 * jpeg_write_scanlines. This lets output of the frame/scan headers be
 * delayed so that application can write COM, etc, markers between
 * jpeg_start_compress and jpeg_write_scanlines.
 */
if (cinfo->master->call_pass_startup)
    (*cinfo->master->pass_startup) (cinfo);

/* Ignore any extra scanlines at bottom of image. */
rows_left = cinfo->image_height - cinfo->next_scanline;
if (num_lines > rows_left)
    num_lines = rows_left;

row_ctr = 0;
(*cinfo->main->process_data) (cinfo, scanlines, &row_ctr, num_lines);
cinfo->next_scanline += row_ctr;
return row_ctr;
}

/*
 * Alternate entry point to write raw data.
 * Processes exactly one iMCU row per call, unless suspended.
 */
GLOBAL(JDIMENSION)
jpeg_write_raw_data (j_compress_ptr cinfo, JSAMPIMAGE data,
                    JDIMENSION num_lines)
{
    JDIMENSION lines_per_iMCU_row;

    if (cinfo->global_state != CSTATE_RAW_OK)
        ERREXIT1(cinfo, JERR_BAD_STATE, cinfo->global_state);
    if (cinfo->next_scanline >= cinfo->image_height) {
        WARNMS(cinfo, JWRN_TOO_MUCH_DATA);
        return 0;
    }

    /* Call progress monitor hook if present */
    if (cinfo->progress != NULL) {
        cinfo->progress->pass_counter = (long) cinfo->next_scanline;
        cinfo->progress->pass_limit = (long) cinfo->image_height;
        (*cinfo->progress->progress_monitor) ((j_common_ptr) cinfo);
    }

    /* Give master control module another chance if this is first call to
     * jpeg_write_raw_data. This lets output of the frame/scan headers be
     * delayed so that application can write COM, etc, markers between
     * jpeg_start_compress and jpeg_write_raw_data.
     */
    if (cinfo->master->call_pass_startup)
        (*cinfo->master->pass_startup) (cinfo);

    /* Verify that at least one iMCU row has been passed. */
    lines_per_iMCU_row = cinfo->max_v_samp_factor * DCTSIZE;
    if (num_lines < lines_per_iMCU_row)
        ERREXIT(cinfo, JERR_BUFFER_SIZE);

    /* Directly compress the row. */
    if (! (*cinfo->coef->compress_data) (cinfo, data)) {
        /* If compressor did not consume the whole row, suspend processing. */
        return 0;
    }

    /* OK, we processed one iMCU row. */
    cinfo->next_scanline += lines_per_iMCU_row;
    return lines_per_iMCU_row;
}

```

[illegible]

```

/*
 * jccoefct.c
 *
 * Copyright (C) 1994-1997, Thomas G. Lane.
 * This file is part of the Independent JPEG Group's software.
 * For conditions of distribution and use, see the accompanying README file.
 *
 * This file contains the coefficient buffer controller for compression.
 * This controller is the top level of the JPEG compressor proper.
 * The coefficient buffer lies between forward-DCT and entropy encoding steps.
 */

#define JPEG_INTERNALS
#include "jinclude.h"
#include "jpeglib.h"

/* We use a full-image coefficient buffer when doing Huffman optimization,
 * and also for writing multiple-scan JPEG files. In all cases, the DCT
 * step is run during the first pass, and subsequent passes need only read
 * the buffered coefficients.
 */
#ifdef ENTROPY_OPT_SUPPORTED
#define FULL_COEF_BUFFER_SUPPORTED
#else
#ifdef C_MULTISCAN_FILES_SUPPORTED
#define FULL_COEF_BUFFER_SUPPORTED
#endif
#endif

/* Private buffer controller object */
typedef struct {
  struct jpeg_c_coef_controller pub; /* public fields */

  JDIMENSION iMCU_row_num; /* iMCU row # within image */
  JDIMENSION mcu_ctr; /* counts MCUs processed in current row */
  int MCU_vert_offset; /* counts MCU rows within iMCU row */
  int MCU_rows_per_iMCU_row; /* number of such rows needed */

  /* For single-pass compression, it's sufficient to buffer just one MCU
   * (although this may prove a bit slow in practice). We allocate a
   * workspace of C_MAX_BLOCKS_IN_MCU coefficient blocks, and reuse it for each
   * MCU constructed and sent. (On 80x86, the workspace is FAR even though
   * it's not really very big; this is to keep the module interfaces unchanged
   * when a large coefficient buffer is necessary.)
   * In multi-pass modes, this array points to the current MCU's blocks
   * within the virtual arrays.
   */
  JBLOCKROW MCU_buffer[C_MAX_BLOCKS_IN_MCU];

  /* In multi-pass modes, we need a virtual block array for each component. */
  jvirt_barray_ptr whole_image[MAX_COMPONENTS];
} my_coef_controller;

typedef my_coef_controller * my_coef_ptr;

/* Forward declarations */
METHODDEF(boolean) compress_data
  (JPP((j_compress_ptr cinfo, JSAMPIMAGE input_buf)));
#ifdef FULL_COEF_BUFFER_SUPPORTED
METHODDEF(boolean) compress_first_pass
  (JPP((j_compress_ptr cinfo, JSAMPIMAGE input_buf)));
METHODDEF(boolean) compress_output
  (JPP((j_compress_ptr cinfo, JSAMPIMAGE input_buf)));
#endif

LOCAL(void)
start_iMCU_row (j_compress_ptr cinfo)
/* Reset within-iMCU-row counters for a new row */
{
  my_coef_ptr coef = (my_coef_ptr) cinfo->coef;

  /* In an interleaved scan, an MCU row is the same as an iMCU row.
   * In a noninterleaved scan, an iMCU row has v_samp_factor MCU rows.
   * But at the bottom of the image, process only what's left.
   */
}

```

```

    if (cinfo->comps_in_scan > 1;
        coef->MCU_rows_per_iMCU_row = 1;
    ) else {
        if (coef->iMCU_row_num < (cinfo->total_iMCU_rows-1))
            coef->MCU_rows_per_iMCU_row = cinfo->cur_comp_info[0]->v_samp_factor;
        else
            coef->MCU_rows_per_iMCU_row = cinfo->cur_comp_info[0]->last_row_height;
    }

    coef->mcu_ctr = 0;
    coef->MCU_vert_offset = 0;
}

/*
 * Initialize for a processing pass.
 */

METHODDEF(void)
start_pass_coef (j_compress_ptr cinfo, J_BUF_MODE pass_mode)
{
    my_coef_ptr coef = (my_coef_ptr) cinfo->coef;

    coef->iMCU_row_num = 0;
    start_iMCU_row(cinfo);

    switch (pass_mode) {
        case JBUF_PASS_THRU:
            if (coef->whole_image[0] != NULL)
                ERREXIT(cinfo, JERR_BAD_BUFFER_MODE);
            coef->pub.compress_data = compress_data;
            break;
#ifdef FULL_COEF_BUFFER_SUPPORTED
        case JBUF_SAVE_AND_PASS:
            if (coef->whole_image[0] == NULL)
                ERREXIT(cinfo, JERR_BAD_BUFFER_MODE);
            coef->pub.compress_data = compress_first_pass;
            break;
        case JBUF_CRANK_DEST:
            if (coef->whole_image[0] == NULL)
                ERREXIT(cinfo, JERR_BAD_BUFFER_MODE);
            coef->pub.compress_data = compress_output;
            break;
#endif
        default:
            ERREXIT(cinfo, JERR_BAD_BUFFER_MODE);
            break;
    }
}

/*
 * Process some data in the single-pass case.
 * We process the equivalent of one fully interleaved MCU row ("iMCU" row)
 * per call, ie, v_samp_factor block rows for each component in the image.
 * Returns TRUE if the iMCU row is completed, FALSE if suspended.
 *
 * NB: input_buf contains a plane for each component in image,
 * which we index according to the component's SOF position.
 */

METHODDEF(boolean)
compress_data (j_compress_ptr cinfo, JSAMPIMAGE input_buf)
{
    my_coef_ptr coef = (my_coef_ptr) cinfo->coef;
    JDIMENSION MCU_col_num; /* index of current MCU within row */
    JDIMENSION last_MCU_col = cinfo->MCUs_per_row - 1;
    JDIMENSION last_iMCU_row = cinfo->total_iMCU_rows - 1;
    int blkcn, bi, ci, yindex, yoffset, blockcnt;
    JDIMENSION ypos, xpos;
    jpeg_component_info *comp_ptr;

    /* Loop to write as much as one whole iMCU row */
    for (yoffset = coef->MCU_vert_offset; yoffset < coef->MCU_rows_per_iMCU_row;
        yoffset++) {
        for (MCU_col_num = coef->mcu_ctr; MCU_col_num <= last_MCU_col;
            MCU_col_num++) {
            /* Determine where data comes from in input_buf and do the DCT thing.
             * Each call on forward_DCT processes a horizontal row of DCT blocks
             * as wide as an MCU; we rely on having allocated the MCU_buffer[] blocks

```

```

* sequentially. Dummy blocks at the right or bottom edge are filled in
* specially. The data in them does not matter for image reconstruction,
* so we fill them with values that will encode to the smallest amount of
* data, viz: all zeroes in the AC entries, DC entries equal to previous
* block's DC value. (Thanks to Thomas Kinsman for this idea.)
*/
blkcn = 0;
for (ci = 0; ci < cinfo->comps_in_scan; ci++) {
  compptr = cinfo->cur_comp_info[ci];
  blockcnt = (MCU_col_num < last_MCU_col) ? compptr->MCU_width
    : compptr->last_col_width;
  xpos = MCU_col_num * compptr->MCU_sample_width;
  ypos = yoffset * DCTSIZE; /* ypos == (yoffset+yindex) * DCTSIZE */
  for (yindex = 0; yindex < compptr->MCU_height; yindex++) {
    if (coef->iMCU_row_num < last_iMCU_row ||
        yoffset+yindex < compptr->last_row_height) {
      (*cinfo->f dct->forward_DCT) (cinfo, compptr,
        input_buf[compptr->component_index],
        coef->MCU_buffer[blkcn],
        ypos, xpos, (JDIMENSION) blockcnt);
      if (blockcnt < compptr->MCU_width) {
        /* Create some dummy blocks at the right edge of the image. */
        jzero_far((void *) coef->MCU_buffer[blkcn + blockcnt],
          (compptr->MCU_width - blockcnt) * SIZEOF(JBLOCK));
        for (bi = blockcnt; bi < compptr->MCU_width; bi++) {
          coef->MCU_buffer[blkcn+bi][0][0] = coef->MCU_buffer[blkcn+bi-1][0][0];
        }
      }
    } else {
      /* Create a row of dummy blocks at the bottom of the image. */
      jzero_far((void *) coef->MCU_buffer[blkcn],
        compptr->MCU_width * SIZEOF(JBLOCK));
      for (bi = 0; bi < compptr->MCU_width; bi++) {
        coef->MCU_buffer[blkcn+bi][0][0] = coef->MCU_buffer[blkcn-1][0][0];
      }
    }
    blkcn += compptr->MCU_width;
    ypos += DCTSIZE;
  }
}
/* Try to write the MCU. In event of a suspension failure, we will
* re-DCT the MCU on restart (a bit inefficient, could be fixed...)
*/
if (!(*cinfo->entropy->encode_mcu) (cinfo, coef->MCU_buffer)) {
  /* Suspension forced; update state counters and exit */
  coef->MCU_vert_offset = yoffset;
  coef->mcu_ctr = MCU_col_num;
  return FALSE;
}
/* Completed an MCU row, but perhaps not an iMCU row */
coef->mcu_ctr = 0;
/* Completed the iMCU row, advance counters for next one */
coef->iMCU_row_num++;
start_iMCU_row(cinfo);
return TRUE;
}

#ifdef FULL_COEF_BUFFER_SUPPORTED

/*
* Process some data in the first pass of a multi-pass case.
* We process the equivalent of one fully interleaved MCU row ("iMCU" row)
* per call, ie, v_samp_factor block rows for each component in the image.
* This amount of data is read from the source buffer, DCT'd and quantized,
* and saved into the virtual arrays. We also generate suitable dummy blocks
* as needed at the right and lower edges. (The dummy blocks are constructed
* in the virtual arrays, which have been padded appropriately.) This makes
* it possible for subsequent passes not to worry about real vs. dummy blocks.
*
* We must also emit the data to the entropy encoder. This is conveniently
* done by calling compress_output() after we've loaded the current strip
* of the virtual arrays.
*
* NB: input_buf contains a plane for each component in image. All
* components are DCT'd and loaded into the virtual arrays in this pass.
* However, it may be that only a subset of the components are emitted to
* the entropy encoder during this first pass; be careful about looking

```

```

* at the scan-dependent variables (MCU dimensions, etc).
*/

```

```

METHODDEF(boolean)
compress_first_pass (j_compress_ptr cinfo, JSAMPIMAGE input_buf)
{
    my_coef_ptr coef = (my_coef_ptr) cinfo->coef;
    JDIMENSION last_iMCU_row = cinfo->total_iMCU_rows - 1;
    JDIMENSION blocks_across, MCUs_across, MCUindex;
    int bi, ci, h_samp_factor, block_row, block_rows, ndummy;
    JCOEF lastDC;
    jpeg_component_info *comp_ptr;
    JBLOCKARRAY buffer;
    JBLOCKROW thisblockrow, lastblockrow;

    for (ci = 0, comp_ptr = cinfo->comp_info; ci < cinfo->num_components;
        ci++, comp_ptr++) {
        /* Align the virtual buffer for this component. */
        buffer = (*cinfo->mem->access_virt_barray)
            ((j_common_ptr) cinfo, coef->whole_image[ci],
             coef->iMCU_row_num * comp_ptr->v_samp_factor,
             (JDIMENSION) comp_ptr->v_samp_factor, TRUE);
        /* Count non-dummy DCT block rows in this iMCU row. */
        if (coef->iMCU_row_num < last_iMCU_row)
            block_rows = comp_ptr->v_samp_factor;
        else {
            /* NB: can't use last_row_height here, since may not be set! */
            block_rows = (int) (comp_ptr->height_in_blocks % comp_ptr->v_samp_factor);
            if (block_rows == 0) block_rows = comp_ptr->v_samp_factor;
        }
        blocks_across = comp_ptr->width_in_blocks;
        h_samp_factor = comp_ptr->h_samp_factor;
        /* Count number of dummy blocks to be added at the right margin. */
        ndummy = (int) (blocks_across % h_samp_factor);
        if (ndummy > 0)
            ndummy = h_samp_factor - ndummy;
        /* Perform DCT for all non-dummy blocks in this iMCU row. Each call
         * on forward_DCT processes a complete horizontal row of DCT blocks.
         */
        for (block_row = 0; block_row < block_rows; block_row++) {
            thisblockrow = buffer[block_row];
            (*cinfo->fdct->forward_DCT) (cinfo, comp_ptr,
                                         input_buf[ci], thisblockrow,
                                         (JDIMENSION) (block_row * DCTSIZE),
                                         (JDIMENSION) 0, blocks_across);
            if (ndummy > 0) {
                /* Create dummy blocks at the right edge of the image. */
                thisblockrow += blocks_across; /* => first dummy block */
                jzero_far((void *) thisblockrow, ndummy * SIZEOF(JBLOCK));
                lastDC = thisblockrow[-1][0];
                for (bi = 0; bi < ndummy; bi++) {
                    thisblockrow[bi][0] = lastDC;
                }
            }
        }
        /* If at end of image, create dummy block rows as needed.
         * The tricky part here is that within each MCU, we want the DC values
         * of the dummy blocks to match the last real block's DC value.
         * This squeezes a few more bytes out of the resulting file...
         */
        if (coef->iMCU_row_num == last_iMCU_row) {
            blocks_across += ndummy; /* include lower right corner */
            MCUs_across = blocks_across / h_samp_factor;
            for (block_row = block_rows; block_row < comp_ptr->v_samp_factor;
                block_row++) {
                thisblockrow = buffer[block_row];
                lastblockrow = buffer[block_row-1];
                jzero_far((void *) thisblockrow,
                         (size_t) (blocks_across * SIZEOF(JBLOCK)));
                for (MCUindex = 0; MCUindex < MCUs_across; MCUindex++) {
                    lastDC = lastblockrow[h_samp_factor-1][0];
                    for (bi = 0; bi < h_samp_factor; bi++) {
                        thisblockrow[bi][0] = lastDC;
                    }
                    thisblockrow += h_samp_factor; /* advance to next MCU in row */
                }
                lastblockrow += h_samp_factor;
            }
        }
    }
}

```



```

/* NB: compress_output will increment iMCU_row_num if successful
 * A suspension return will result in redoing all the work above at time.
 */

/* Emit data to the entropy encoder, sharing code with subsequent passes */
return compress_output(cinfo, input_buf);
}

/*
 * Process some data in subsequent passes of a multi-pass case.
 * We process the equivalent of one fully interleaved MCU row ("iMCU" row)
 * per call, ie, v_samp_factor block rows for each component in the scan.
 * The data is obtained from the virtual arrays and fed to the entropy coder.
 * Returns TRUE if the iMCU row is completed, FALSE if suspended.
 *
 * NB: input_buf is ignored; it is likely to be a NULL pointer.
 */

METHODDEF(boolean)
compress_output (j_compress_ptr cinfo, JSAMPIMAGE input_buf)
{
    my_coef_ptr coef = (my_coef_ptr) cinfo->coef;
    JDIMENSION MCU_col_num; /* index of current MCU within row */
    int blkcn, ci, xindex, yindex, yoffset;
    JDIMENSION start_col;
    JBLOCKARRAY buffer[MAX_COMPS_IN_SCAN];
    JBLOCKROW buffer_ptr;
    jpeg_component_info *comp_ptr;

    /* Align the virtual buffers for the components used in this scan.
     * NB: during first pass, this is safe only because the buffers will
     * already be aligned properly, so jmemmgr.c won't need to do any I/O.
     */
    for (ci = 0; ci < cinfo->comps_in_scan; ci++) {
        comp_ptr = cinfo->cur_comp_info[ci];
        buffer[ci] = (*cinfo->mem->access_virt_barray)
            ((j_common_ptr) cinfo, coef->whole_image[comp_ptr->component_index],
             coef->iMCU_row_num * comp_ptr->v_samp_factor,
             (JDIMENSION) comp_ptr->v_samp_factor, FALSE);
    }

    /* Loop to process one whole iMCU row */
    for (yoffset = coef->MCU_vert_offset; yoffset < coef->MCU_rows_per_iMCU_row;
         yoffset++) {
        for (MCU_col_num = coef->mcu_ctr; MCU_col_num < cinfo->MCUs_per_row;
             MCU_col_num++) {
            /* Construct list of pointers to DCT blocks belonging to this MCU */
            blkcn = 0; /* index of current DCT block within MCU */
            for (ci = 0; ci < cinfo->comps_in_scan; ci++) {
                comp_ptr = cinfo->cur_comp_info[ci];
                start_col = MCU_col_num * comp_ptr->MCU_width;
                for (yindex = 0; yindex < comp_ptr->MCU_height; yindex++) {
                    buffer_ptr = buffer[ci][yindex+yoffset] + start_col;
                    for (xindex = 0; xindex < comp_ptr->MCU_width; xindex++) {
                        coef->MCU_buffer[blkcn++] = buffer_ptr++;
                    }
                }
            }
            /* Try to write the MCU. */
            if (! (*cinfo->entropy->encode_mcu) (cinfo, coef->MCU_buffer)) {
                /* Suspension forced; update state counters and exit */
                coef->MCU_vert_offset = yoffset;
                coef->mcu_ctr = MCU_col_num;
                return FALSE;
            }
        }
        /* Completed an MCU row, but perhaps not an iMCU row */
        coef->mcu_ctr = 0;
    }
    /* Completed the iMCU row, advance counters for next one */
    coef->iMCU_row_num++;
    start_iMCU_row(cinfo);
    return TRUE;
}

#endif /* FULL_COEF_BUFFER_SUPPORTED */

/*

```

```

* Initialize coefficient buffer controller.
*/

GLOBAL(void)
jinit_c_coef_controller (j_compress_ptr cinfo, boolean need_full_buffer)
{
    my_coef_ptr coef;

    coef = (my_coef_ptr)
        (*cinfo->mem->alloc_small) ((j_common_ptr) cinfo, JPOOL_IMAGE,
            sizeof(my_coef_controller));
    cinfo->coef = (struct jpeg_c_coef_controller *) coef;
    coef->pub.start_pass = start_pass_coef;

    /* Create the coefficient buffer. */
    if (need_full_buffer) {
#ifdef FULL_COEF_BUFFER_SUPPORTED
        /* Allocate a full-image virtual array for each component, */
        /* padded to a multiple of samp_factor DCT blocks in each direction. */
        int ci;
        jpeg_component_info *comp_ptr;

        for (ci = 0, comp_ptr = cinfo->comp_info; ci < cinfo->num_components;
            ci++, comp_ptr++) {
            coef->whole_image[ci] = (*cinfo->mem->request_virt_barray)
                ((j_common_ptr) cinfo, JPOOL_IMAGE, FALSE,
                (JDIMENSION) jround_up((long) comp_ptr->width_in_blocks,
                    (long) comp_ptr->h_samp_factor),
                (JDIMENSION) jround_up((long) comp_ptr->height_in_blocks,
                    (long) comp_ptr->v_samp_factor),
                (JDIMENSION) comp_ptr->v_samp_factor);
        }
    }
    else
        ERREXIT(cinfo, JERR_BAD_BUFFER_MODE);
#endif
    else {
        /* We only need a single-MCU buffer. */
        JBLOCKROW buffer;
        int i;

        buffer = (JBLOCKROW)
            (*cinfo->mem->alloc_large) ((j_common_ptr) cinfo, JPOOL_IMAGE,
                C_MAX_BLOCKS_IN_MCU * sizeof(JBLOCK));
        for (i = 0; i < C_MAX_BLOCKS_IN_MCU; i++) {
            coef->MCU_buffer[i] = buffer + i;
        }
        coef->whole_image[0] = NULL; /* flag for no virtual arrays */
    }
}

```

```

/*
 * jccolor.c
 *
 * Copyright (C) 1991-1996, Thomas G. Lane.
 * This file is part of the Independent JPEG Group's software.
 * For conditions of distribution and use, see the accompanying README file.
 *
 * This file contains input colorspace conversion routines.
 */

#define JPEG_INTERNALS
#include "jinclude.h"
#include "jpeglib.h"

/* Private subobject */

typedef struct {
  struct jpeg_color_converter pub; /* public fields */

  /* Private state for RGB->YCC conversion */
  INT32 * rgb_ycc_tab; /* => table for RGB to YCbCr conversion */
} my_color_converter;

typedef my_color_converter * my_cconvert_ptr;

/***** RGB -> YCbCr conversion: most common case *****/

/*
 * YCbCr is defined per CCIR 601-1, except that Cb and Cr are
 * normalized to the range 0..MAXJSAMPLE rather than -0.5 .. 0.5.
 * The conversion equations to be implemented are therefore
 * 
$$Y = 0.29900 * R + 0.58700 * G + 0.11400 * B$$

 * 
$$Cb = -0.16874 * R - 0.33126 * G + 0.50000 * B + CENTERJSAMPLE$$

 * 
$$Cr = 0.50000 * R - 0.41869 * G - 0.08131 * B + CENTERJSAMPLE$$

 * (These numbers are derived from TIFF 6.0 section 21, dated 3-June-92.)
 * Note: older versions of the IJG code used a zero offset of MAXJSAMPLE/2,
 * rather than CENTERJSAMPLE, for Cb and Cr. This gave equal positive and
 * negative swings for Cb/Cr, but meant that grayscale values (Cb=Cr=0)
 * were not represented exactly. Now we sacrifice exact representation of
 * maximum red and maximum blue in order to get exact grayscales.
 *
 * To avoid floating-point arithmetic, we represent the fractional constants
 * as integers scaled up by 2^16 (about 4 digits precision); we have to divide
 * the products by 2^16, with appropriate rounding, to get the correct answer.
 *
 * For even more speed, we avoid doing any multiplications in the inner loop
 * by precalculating the constants times R,G,B for all possible values.
 * For 8-bit JSAMPLEs this is very reasonable (only 256 entries per table);
 * for 12-bit samples it is still acceptable. It's not very reasonable for
 * 16-bit samples, but if you want lossless storage you shouldn't be changing
 * colorspace anyway.
 * The CENTERJSAMPLE offsets and the rounding fudge-factor of 0.5 are included
 * in the tables to save adding them separately in the inner loop.
 */

#define SCALEBITS 16 /* speediest right-shift on some machines */
#define CBR_OFFSET ((INT32) CENTERJSAMPLE << SCALEBITS)
#define ONE_HALF ((INT32) 1 << (SCALEBITS-1))
#define FIX(x) ((INT32) ((x) * (1L<<SCALEBITS) + 0.5))

/*
 * We allocate one big table and divide it up into eight parts, instead of
 * doing eight alloc_small requests. This lets us use a single table base
 * address, which can be held in a register in the inner loops on many
 * machines (more than can hold all eight addresses, anyway).
 */

#define R_Y_OFF 0 /* offset to R => Y section */
#define G_Y_OFF (1*(MAXJSAMPLE+1)) /* offset to G => Y section */
#define B_Y_OFF (2*(MAXJSAMPLE+1)) /* etc. */
#define R_CB_OFF (3*(MAXJSAMPLE+1))
#define G_CB_OFF (4*(MAXJSAMPLE+1))
#define B_CB_OFF (5*(MAXJSAMPLE+1))
#define R_CR_OFF B_CB_OFF /* B=>Cb, R=>Cr are the same */
#define G_CR_OFF (6*(MAXJSAMPLE+1))
#define B_CR_OFF (7*(MAXJSAMPLE+1))
#define TABLE_SIZE (8*(MAXJSAMPLE+1))

```

```

/*
 * Initialize for RGB->YCC color space conversion.
 */

METHODDEF(void)
rgb_ycc_start (j_compress_ptr cinfo)
{
    my_cconvert_ptr cconvert = (my_cconvert_ptr) cinfo->cconvert;
    INT32 * rgb_ycc_tab;
    INT32 i;

    /* Allocate and fill in the conversion tables. */
    cconvert->rgb_ycc_tab = rgb_ycc_tab = (INT32 *)
        (*cinfo->mem->alloc_small) ((j_common_ptr) cinfo, JPOOL_IMAGE,
            (TABLE_SIZE * SIZEOF(INT32)));

    for (i = 0; i <= MAXJSAMPLE; i++) {
        rgb_ycc_tab[i+R_Y_OFF] = FIX(0.29900) * i;
        rgb_ycc_tab[i+G_Y_OFF] = FIX(0.58700) * i;
        rgb_ycc_tab[i+B_Y_OFF] = FIX(0.11400) * i + ONE_HALF;
        rgb_ycc_tab[i+R_CB_OFF] = (-FIX(0.16874)) * i;
        rgb_ycc_tab[i+G_CB_OFF] = (-FIX(0.33126)) * i;
        /* We use a rounding fudge-factor of 0.5-epsilon for Cb and Cr.
         * This ensures that the maximum output will round to MAXJSAMPLE
         * not MAXJSAMPLE+1, and thus that we don't have to range-limit.
         */
        rgb_ycc_tab[i+B_CB_OFF] = FIX(0.50000) * i + CBCR_OFFSET + ONE_HALF-1;
    /* B->Cb and R->Cr tables are the same
    rgb_ycc_tab[i+R_CR_OFF] = FIX(0.50000) * i + CBCR_OFFSET + ONE_HALF-1;
    */
        rgb_ycc_tab[i+G_CR_OFF] = (-FIX(0.41869)) * i;
        rgb_ycc_tab[i+B_CR_OFF] = (-FIX(0.08131)) * i;
    }
}

/*
 * Convert some rows of samples to the JPEG colorspace.
 *
 * Note that we change from the application's interleaved-pixel format
 * to our internal noninterleaved, one-plane-per-component format.
 * The input buffer is therefore three times as wide as the output buffer.
 *
 * A starting row offset is provided only for the output buffer. The caller
 * can easily adjust the passed input_buf value to accommodate any row
 * offset required on that side.
 */

METHODDEF(void)
rgb_ycc_convert (j_compress_ptr cinfo,
    JSAMPARRAY input_buf, JSAMPIMAGE output_buf,
    JDIMENSION output_row, int num_rows)
{
    my_cconvert_ptr cconvert = (my_cconvert_ptr) cinfo->cconvert;
    register int r, g, b;
    register INT32 * ctab = cconvert->rgb_ycc_tab;
    register JSAMPROW inptr;
    register JSAMPROW outptr0, outptr1, outptr2;
    register JDIMENSION col;
    JDIMENSION num_cols = cinfo->image_width;

    while (--num_rows >= 0) {
        inptr = *input_buf++;
        outptr0 = output_buf[0][output_row];
        outptr1 = output_buf[1][output_row];
        outptr2 = output_buf[2][output_row];
        output_row++;
        for (col = 0; col < num_cols; col++) {
            r = GETJSAMPLE(inptr[RGB_RED]);
            g = GETJSAMPLE(inptr[RGB_GREEN]);
            b = GETJSAMPLE(inptr[RGB_BLUE]);
            inptr += RGB_PIXELSIZE;
            /* If the inputs are 0..MAXJSAMPLE, the outputs of these equations
             * must be too; we do not need an explicit range-limiting operation.
             * Hence the value being shifted is never negative, and we don't
             * need the general RIGHT_SHIFT macro.
             */
            /* Y */
            outptr0[col] = (JSAMPLE)
                ((ctab[r+R_Y_OFF] + ctab[g+G_Y_OFF] + ctab[b+B_Y_OFF])

```

```

        >> SCALEBITS);
/* Cb */
outptr1[col] = (JSAMPLE)
    ((ctab[r+R_CB_OFF] + ctab[g+G_CB_OFF] + ctab[b+B_CB_OFF])
    >> SCALEBITS);
/* Cr */
outptr2[col] = (JSAMPLE)
    ((ctab[r+R_CR_OFF] + ctab[g+G_CR_OFF] + ctab[b+B_CR_OFF])
    >> SCALEBITS);
    }
}

/***** Cases other than RGB -> YCbCr *****/

/*
 * Convert some rows of samples to the JPEG colorspace.
 * This version handles RGB->grayscale conversion, which is the same
 * as the RGB->Y portion of RGB->YCbCr.
 * We assume rgb_ycc_start has been called (we only use the Y tables).
 */

METHODDEF(void)
rgb_gray_convert (j_compress_ptr cinfo,
                  JSAMPARRAY input_buf, JSAMPIMAGE output_buf,
                  JDIMENSION output_row, int num_rows)
{
    my_cconvert_ptr cconvert = (my_cconvert_ptr) cinfo->cconvert;
    register int r, g, b;
    register INT32 * ctab = cconvert->rgb_ycc_tab;
    register JSAMPROW inptr;
    register JSAMPROW outptr;
    register JDIMENSION col;
    JDIMENSION num_cols = cinfo->image_width;

    while (--num_rows >= 0) {
        inptr = *input_buf++;
        outptr = output_buf[0][output_row];
        output_row++;
        for (col = 0; col < num_cols; col++) {
            r = GETJSAMPLE(inptr[RGB_RED]);
            g = GETJSAMPLE(inptr[RGB_GREEN]);
            b = GETJSAMPLE(inptr[RGB_BLUE]);
            inptr += RGB_PIXELSIZE;
            /* Y */
            outptr[col] = (JSAMPLE)
                ((ctab[r+R_Y_OFF] + ctab[g+G_Y_OFF] + ctab[b+B_Y_OFF])
                >> SCALEBITS);
        }
    }
}

/*
 * Convert some rows of samples to the JPEG colorspace.
 * This version handles Adobe-style CMYK->YCCK conversion,
 * where we convert R=1-C, G=1-M, and B=1-Y to YCbCr using the same
 * conversion as above, while passing K (black) unchanged.
 * We assume rgb_ycc_start has been called.
 */

```

```

METHODDEF(void)
cmyk_ycck_convert (j_compress_ptr cinfo,
                   JSAMPARRAY input_buf, JSAMPIMAGE output_buf,
                   JDIMENSION output_row, int num_rows)
{
    my_cconvert_ptr cconvert = (my_cconvert_ptr) cinfo->cconvert;
    register int r, g, b;
    register INT32 * ctab = cconvert->rgb_ycc_tab;
    register JSAMPROW inptr;
    register JSAMPROW outptr0, outptr1, outptr2, outptr3;
    register JDIMENSION col;
    JDIMENSION num_cols = cinfo->image_width;

    while (--num_rows >= 0) {
        inptr = *input_buf++;
        outptr0 = output_buf[0][output_row];
        outptr1 = output_buf[1][output_row];
    }
}

```

```

outptr2 = output_buf[2][output_row];
outptr3 = output_buf[3][output_row];
output_row++;
for (col = 0; col < num_cols; col++) {
    r = MAXJSAMPLE - GETJSAMPLE(inptr[0]);
    g = MAXJSAMPLE - GETJSAMPLE(inptr[1]);
    b = MAXJSAMPLE - GETJSAMPLE(inptr[2]);
    /* K passes through as-is */
    outptr3[col] = inptr[3]; /* don't need GETJSAMPLE here */
    inptr += 4;
    /* If the inputs are 0..MAXJSAMPLE, the outputs of these equations
     * must be too; we do not need an explicit range-limiting operation.
     * Hence the value being shifted is never negative, and we don't
     * need the general RIGHT_SHIFT macro.
     */
    /* Y */
    outptr0[col] = (JSAMPLE)
        ((ctab[r+R_Y_OFF] + ctab[g+G_Y_OFF] + ctab[b+B_Y_OFF])
         >> SCALEBITS);
    /* Cb */
    outptr1[col] = (JSAMPLE)
        ((ctab[r+R_CB_OFF] + ctab[g+G_CB_OFF] + ctab[b+B_CB_OFF])
         >> SCALEBITS);
    /* Cr */
    outptr2[col] = (JSAMPLE)
        ((ctab[r+R_CR_OFF] + ctab[g+G_CR_OFF] + ctab[b+B_CR_OFF])
         >> SCALEBITS);
}
}
}

```

\* Convert some rows of samples to the JPEG colorspace.  
 \* This version handles grayscale output with no conversion.  
 \* The source can be either plain grayscale or YCbCr (since Y == gray).

```

METHODDEF(void)
grayscale_convert (j_compress_ptr cinfo,
                  JSAMPARRAY input_buf, JSAMPIMAGE output_buf,
                  JDIMENSION output_row, int num_rows)
{
    register JSAMPROW inptr;
    register JSAMPROW outptr;
    register JDIMENSION col;
    JDIMENSION num_cols = cinfo->image_width;
    int instride = cinfo->input_components;

    while (--num_rows >= 0) {
        inptr = *input_buf++;
        outptr = output_buf[0][output_row];
        output_row++;
        for (col = 0; col < num_cols; col++) {
            outptr[col] = inptr[0]; /* don't need GETJSAMPLE() here */
            inptr += instride;
        }
    }
}

```

/\*  
 \* Convert some rows of samples to the JPEG colorspace.  
 \* This version handles multi-component colorspace without conversion.  
 \* We assume input\_components == num\_components.  
 \*/

```

METHODDEF(void)
null_convert (j_compress_ptr cinfo,
              JSAMPARRAY input_buf, JSAMPIMAGE output_buf,
              JDIMENSION output_row, int num_rows)
{
    register JSAMPROW inptr;
    register JSAMPROW outptr;
    register JDIMENSION col;
    register int ci;
    int nc = cinfo->num_components;
    JDIMENSION num_cols = cinfo->image_width;

    while (--num_rows >= 0) {

```

```

/* It seems fastest to make a separate pass for each component
for (ci = 0; ci < nc; ci++) {
    inptr = *input_buf;
    outptr = output_buf[ci][output_row];
    for (col = 0; col < num_cols; col++) {
        outptr[col] = inptr[ci]; /* don't need GETJSAMPLE() here */
        inptr += nc;
    }
    input_buf++;
    output_row++;
}
}

/*
 * Empty method for start_pass.
 */

METHODDEF(void)
null_method (j_compress_ptr cinfo)
{
    /* no work needed */
}

/*
 * Module initialization routine for input colorspace conversion.
 */

GLOBAL(void)
jinit_color_converter (j_compress_ptr cinfo)
{
    my_cconvert_ptr cconvert;
    cconvert = (my_cconvert_ptr)
        (*cinfo->mem->alloc_small) ((j_common_ptr) cinfo, JPOOL_IMAGE,
        sizeof(my_color_converter));
    cinfo->cconvert = (struct jpeg_color_converter *) cconvert;
    /* set start_pass to null method until we find out differently */
    cconvert->pub.start_pass = null_method;

    /* Make sure input_components agrees with in_color_space */
    switch (cinfo->in_color_space) {
        case JCS_GRAYSCALE:
            if (cinfo->input_components != 1)
                ERREXIT(cinfo, JERR_BAD_IN_COLORSPACE);
            break;
        case JCS_RGB:
            #if RGB_PIXELSIZE != 3
            if (cinfo->input_components != RGB_PIXELSIZE)
                ERREXIT(cinfo, JERR_BAD_IN_COLORSPACE);
            break;
            #endif /* else share code with YCbCr */

        case JCS_YCbCr:
            if (cinfo->input_components != 3)
                ERREXIT(cinfo, JERR_BAD_IN_COLORSPACE);
            break;

        case JCS_CMYK:
        case JCS_YCCK:
            if (cinfo->input_components != 4)
                ERREXIT(cinfo, JERR_BAD_IN_COLORSPACE);
            break;

        default:
            /* JCS_UNKNOWN can be anything */
            if (cinfo->input_components < 1)
                ERREXIT(cinfo, JERR_BAD_IN_COLORSPACE);
            break;
    }

    /* Check num_components, set conversion method based on requested space */
    switch (cinfo->jpeg_color_space) {
        case JCS_GRAYSCALE:
            if (cinfo->num_components != 1)
                ERREXIT(cinfo, JERR_BAD_J_COLORSPACE);
            if (cinfo->in_color_space == JCS_GRAYSCALE)
                cconvert->pub.color_convert = grayscale_convert;
    }

```

```

else if (cinfo->in_color_space == JCS_RGB) {
    cconvert->pub.start_pass = rgb_ycc_start;
    cconvert->pub.color_convert = rgb_gray_convert;
} else if (cinfo->in_color_space == JCS_YCbCr)
    cconvert->pub.color_convert = grayscale_convert;
else
    ERREXIT(cinfo, JERR_CONVERSION_NOTIMPL);
break;

case JCS_RGB:
    if (cinfo->num_components != 3)
        ERREXIT(cinfo, JERR_BAD_J_COLORSPACE);
    if (cinfo->in_color_space == JCS_RGB && RGB_PIXELSIZE == 3)
        cconvert->pub.color_convert = null_convert;
    else
        ERREXIT(cinfo, JERR_CONVERSION_NOTIMPL);
    break;

case JCS_YCbCr:
    if (cinfo->num_components != 3)
        ERREXIT(cinfo, JERR_BAD_J_COLORSPACE);
    if (cinfo->in_color_space == JCS_RGB) {
        cconvert->pub.start_pass = rgb_ycc_start;
        cconvert->pub.color_convert = rgb_ycc_convert;
    } else if (cinfo->in_color_space == JCS_YCbCr)
        cconvert->pub.color_convert = null_convert;
    else
        ERREXIT(cinfo, JERR_CONVERSION_NOTIMPL);
    break;

case JCS_CMYK:
    if (cinfo->num_components != 4)
        ERREXIT(cinfo, JERR_BAD_J_COLORSPACE);
    if (cinfo->in_color_space == JCS_CMYK)
        cconvert->pub.color_convert = null_convert;
    else
        ERREXIT(cinfo, JERR_CONVERSION_NOTIMPL);
    break;

case JCS_YCCK:
    if (cinfo->num_components != 4)
        ERREXIT(cinfo, JERR_BAD_J_COLORSPACE);
    if (cinfo->in_color_space == JCS_CMYK) {
        cconvert->pub.start_pass = rgb_ycc_start;
        cconvert->pub.color_convert = cmyk_ycck_convert;
    } else if (cinfo->in_color_space == JCS_YCCK)
        cconvert->pub.color_convert = null_convert;
    else
        ERREXIT(cinfo, JERR_CONVERSION_NOTIMPL);
    break;

default:
    /* allow null conversion of JCS_UNKNOWN */
    if (cinfo->jpeg_color_space != cinfo->in_color_space ||
        cinfo->num_components != cinfo->input_components)
        ERREXIT(cinfo, JERR_CONVERSION_NOTIMPL);
    cconvert->pub.color_convert = null_convert;
    break;
}
}

```



```

/*
 * jcdctmgr.c
 *
 * Copyright (C) 1994-1996, Thomas G. Lane.
 * This file is part of the Independent JPEG Group's software.
 * For conditions of distribution and use, see the accompanying README file.
 *
 * This file contains the forward-DCT management logic.
 * This code selects a particular DCT implementation to be used,
 * and it performs related housekeeping chores including coefficient
 * quantization.
 */

#define JPEG_INTERNALS
#include "jinclude.h"
#include "jpeglib.h"
#include "jdct.h"          /* Private declarations for DCT subsystem */

/* Private subobject for this module */

typedef struct {
  struct jpeg_forward_dct pub; /* public fields */

  /* Pointer to the DCT routine actually in use */
  forward_DCT_method_ptr do_dct;

  /* The actual post-DCT divisors --- not identical to the quant table
   * entries, because of scaling (especially for an unnormalized DCT).
   * Each table is given in normal array order.
   */
  DCTELEM * divisors[NUM_QUANT_TBLS];

#ifdef DCT_FLOAT_SUPPORTED
  /* Same as above for the floating-point case. */
  float_DCT_method_ptr do_float_dct;
  FAST_FLOAT * float_divisors[NUM_QUANT_TBLS];
#endif
} my_fdct_controller;

typedef my_fdct_controller * my_fdct_ptr;

/*
 * Initialize for a processing pass.
 * Verify that all referenced Q-tables are present, and set up
 * the divisor table for each one.
 * In the current implementation, DCT of all components is done during
 * the first pass, even if only some components will be output in the
 * first scan. Hence all components should be examined here.
 */
METHODDEF(void)
start_pass_fdctmgr (j_compress_ptr cinfo)
{
  my_fdct_ptr fdct = (my_fdct_ptr) cinfo->fdct;
  int ci, qtblno, i;
  jpeg_component_info *comp_ptr;
  JQUANT_TBL * qtbl;
  DCTELEM * dtbl;

  for (ci = 0, comp_ptr = cinfo->comp_info; ci < cinfo->num_components;
       ci++, comp_ptr++) {
    qtblno = comp_ptr->quant_tbl_no;
    /* Make sure specified quantization table is present */
    if (qtblno < 0 || qtblno >= NUM_QUANT_TBLS ||
        cinfo->quant_tbl_ptrs[qtblno] == NULL)
      ERREXIT1(cinfo, JERR_NO_QUANT_TABLE, qtblno);
    qtbl = cinfo->quant_tbl_ptrs[qtblno];
    /* Compute divisors for this quant table */
    /* We may do this more than once for same table, but it's not a big deal */
    switch (cinfo->dct_method) {
#ifdef DCT_ISLOW_SUPPORTED
    case JDCT_ISLOW:
      /* For LL&M IDCT method, divisors are equal to raw quantization
       * coefficients multiplied by 8 (to counteract scaling).
       */
      if (fdct->divisors[qtblno] == NULL) {
        fdct->divisors[qtblno] = (DCTELEM *)
          (*cinfo->mem->alloc_small) ((j_common_ptr) cinfo, JPOOL_IMAGE,

```

```

        DCTSIZE2 * sizeof(DCTELEM));
    }
    dtbl = fdct->divisors[qtblno];
    for (i = 0; i < DCTSIZE2; i++) {
        dtbl[i] = ((DCTELEM) qtbl->quantval[i]) << 3;
    }
    break;
#endif
#ifdef DCT_IFAST_SUPPORTED
case JDCT_IFAST:
{
    /* For AA&N IDCT method, divisors are equal to quantization
     * coefficients scaled by scalefactor[row]*scalefactor[col], where
     *   scalefactor[0] = 1
     *   scalefactor[k] = cos(k*PI/16) * sqrt(2)    for k=1..7
     * We apply a further scale factor of 8.
     */
#define CONST_BITS 14
    static const INT16 aanscales[DCTSIZE2] = {
        /* precomputed values scaled up by 14 bits */
        16384, 22725, 21407, 19266, 16384, 12873, 8867, 4520,
        22725, 31521, 29692, 26722, 22725, 17855, 12299, 6270,
        21407, 29692, 27969, 25172, 21407, 16819, 11585, 5906,
        19266, 26722, 25172, 22654, 19266, 15137, 10426, 5315,
        16384, 22725, 21407, 19266, 16384, 12873, 8867, 4520,
        12873, 17855, 16819, 15137, 12873, 10114, 6967, 3552,
        8867, 12299, 11585, 10426, 8867, 6967, 4799, 2446,
        4520, 6270, 5906, 5315, 4520, 3552, 2446, 1247
    };
    SHIFT_TEMPS

    if (fdct->divisors[qtblno] == NULL) {
        fdct->divisors[qtblno] = (DCTELEM *)
            (*cinfo->mem->alloc_small) ((j_common_ptr) cinfo, JPOOL_IMAGE,
                DCTSIZE2 * sizeof(DCTELEM));
    }
    dtbl = fdct->divisors[qtblno];
    for (i = 0; i < DCTSIZE2; i++) {
        dtbl[i] = (DCTELEM)
            DESCALE(MULTIPLY16V16((INT32) qtbl->quantval[i],
                (INT32) aanscales[i]),
                CONST_BITS-3);
    }
    break;
#endif
#ifdef DCT_FLOAT_SUPPORTED
case JDCT_FLOAT:
{
    /* For float AA&N IDCT method, divisors are equal to quantization
     * coefficients scaled by scalefactor[row]*scalefactor[col], where
     *   scalefactor[0] = 1
     *   scalefactor[k] = cos(k*PI/16) * sqrt(2)    for k=1..7
     * We apply a further scale factor of 8.
     * What's actually stored is 1/divisor so that the inner loop can
     * use a multiplication rather than a division.
     */
    FAST_FLOAT * fdtbl;
    int row, col;
    static const double aanscalefactor[DCTSIZE] = {
        1.0, 1.387039845, 1.306562965, 1.175875602,
        1.0, 0.785694958, 0.541196100, 0.275899379
    };

    if (fdct->float_divisors[qtblno] == NULL) {
        fdct->float_divisors[qtblno] = (FAST_FLOAT *)
            (*cinfo->mem->alloc_small) ((j_common_ptr) cinfo, JPOOL_IMAGE,
                DCTSIZE2 * sizeof(FAST_FLOAT));
    }
    fdtbl = fdct->float_divisors[qtblno];
    i = 0;
    for (row = 0; row < DCTSIZE; row++) {
        for (col = 0; col < DCTSIZE; col++) {
            fdtbl[i] = (FAST_FLOAT)
                (1.0 / (((double) qtbl->quantval[i] *
                    aanscalefactor[row] * aanscalefactor[col] * 8.0)));
            i++;
        }
    }
}
#endif
}

```

```

        break;
#endif
    default:
        ERREXIT(cinfo, JERR_NOT_COMPILED);
        break;
    }
}

/*
 * Perform forward DCT on one or more blocks of a component.
 *
 * The input samples are taken from the sample_data[] array starting at
 * position start_row/start_col, and moving to the right for any additional
 * blocks. The quantized coefficients are returned in coef_blocks[].
 */

METHODDEF(void)
forward_DCT (j_compress_ptr cinfo, jpeg_component_info * comptr,
             JSAMPARRAY sample_data, JBLOCKROW coef_blocks,
             JDIMENSION start_row, JDIMENSION start_col,
             JDIMENSION num_blocks)
/* This version is used for integer DCT implementations. */
{
    /* This routine is heavily used, so it's worth coding it tightly. */
    my_fdct_ptr fdct = (my_fdct_ptr) cinfo->fdct;
    forward_DCT_method_ptr do_dct = fdct->do_dct;
    DCTELEM * divisors = fdct->divisors[comptr->quant_tbl_no];
    DCTELEM workspace[DCTSIZE2]; /* work area for FDCT subroutine */
    JDIMENSION bi;

    sample_data += start_row; /* fold in the vertical offset once */

    for (bi = 0; bi < num_blocks; bi++, start_col += DCTSIZE) {
        /* Load data into workspace, applying unsigned->signed conversion */
        { register DCTELEM *workspaceptr;
          register JSAMPROW elem_ptr;
          register int elemr;

          workspaceptr = workspace;
          for (elemr = 0; elemr < DCTSIZE; elemr++) {
              elem_ptr = sample_data[elemr] + start_col;
              #if DCTSIZE == 8 /* unroll the inner loop */
                  *workspaceptr++ = GETJSAMPLE(*elem_ptr++) - CENTERJSAMPLE;
                  *workspaceptr++ = GETJSAMPLE(*elem_ptr++) - CENTERJSAMPLE;
                  *workspaceptr++ = GETJSAMPLE(*elem_ptr++) - CENTERJSAMPLE;
                  *workspaceptr++ = GETJSAMPLE(*elem_ptr++) - CENTERJSAMPLE;
                  *workspaceptr++ = GETJSAMPLE(*elem_ptr++) - CENTERJSAMPLE;
                  *workspaceptr++ = GETJSAMPLE(*elem_ptr++) - CENTERJSAMPLE;
                  *workspaceptr++ = GETJSAMPLE(*elem_ptr++) - CENTERJSAMPLE;
                  *workspaceptr++ = GETJSAMPLE(*elem_ptr++) - CENTERJSAMPLE;
              #else
                  { register int elemc;
                    for (elemc = DCTSIZE; elemc > 0; elemc--) {
                        *workspaceptr++ = GETJSAMPLE(*elem_ptr++) - CENTERJSAMPLE;
                    }
                }
              #endif
          }
        }
        #endif
    }

    /* Perform the DCT */
    (*do_dct) (workspace);

    /* Quantize/descale the coefficients, and store into coef_blocks[] */
    { register DCTELEM temp, qval;
      register int i;
      register JOEFPTR output_ptr = coef_blocks[bi];

      for (i = 0; i < DCTSIZE2; i++) {
          qval = divisors[i];
          temp = workspace[i];
          /* Divide the coefficient value by qval, ensuring proper rounding.
           * Since C does not specify the direction of rounding for negative
           * quotients, we have to force the dividend positive for portability.
           *
           * In most files, at least half of the output values will be zero
           * (at default quantization settings, more like three-quarters...)
           * so we should ensure that this case is fast. On many machines,

```

```

* a comparison is enough faster than a divide to make a speed test
* a win. Since both inputs will be nonnegative, we need only test
* for a < b to discover whether a/b is 0.
* If your machine's division is fast enough, define FAST_DIVIDE.
*/
#ifdef FAST_DIVIDE
#define DIVIDE_BY(a,b)  a /= b
#else
#define DIVIDE_BY(a,b)  if (a >= b) a /= b; else a = 0
#endif
    if (temp < 0) {
        temp = -temp;
        temp += qval>>1; /* for rounding */
        DIVIDE_BY(temp, qval);
        temp = -temp;
    } else {
        temp += qval>>1; /* for rounding */
        DIVIDE_BY(temp, qval);
    }
    output_ptr[i] = (JCOEF) temp;
}
}

#ifdef DCT_FLOAT_SUPPORTED

METHODDEF(void)
forward_DCT_float (j_compress_ptr cinfo, jpeg_component_info * comp_ptr,
    JSAMPARRAY sample_data, JBLOCKROW coef_blocks,
    JDIMENSION start_row, JDIMENSION start_col,
    JDIMENSION num_blocks)
/* This version is used for floating-point DCT implementations. */
{
    /* This routine is heavily used, so it's worth coding it tightly. */
    my_fdct_ptr fdct = (my_fdct_ptr) cinfo->fdct;
    float_DCT_method_ptr do_dct = fdct->do_float_dct;
    FAST_FLOAT * divisors = fdct->float_divisors[comp_ptr->quant_tbl_no];
    FAST_FLOAT workspace[DCTSIZE2]; /* work area for FDCT subroutine */
    JDIMENSION bi;

    sample_data += start_row; /* fold in the vertical offset once */

    for (bi = 0; bi < num_blocks; bi++, start_col += DCTSIZE) {
        /* Load data into workspace, applying unsigned->signed conversion */
        { register FAST_FLOAT *workspaceptr;
          register JSAMPROW elem_ptr;
          register int elem_r;

          workspaceptr = workspace;
          for (elem_r = 0; elem_r < DCTSIZE; elem_r++) {
              elem_ptr = sample_data[elem_r] + start_col;
#ifdef DCTSIZE == 8 /* unroll the inner loop */
              *workspaceptr++ = (FAST_FLOAT)(GETJSAMPLE(*elem_ptr++) - CENTERJSAMPLE);
              *workspaceptr++ = (FAST_FLOAT)(GETJSAMPLE(*elem_ptr++) - CENTERJSAMPLE);
              *workspaceptr++ = (FAST_FLOAT)(GETJSAMPLE(*elem_ptr++) - CENTERJSAMPLE);
              *workspaceptr++ = (FAST_FLOAT)(GETJSAMPLE(*elem_ptr++) - CENTERJSAMPLE);
              *workspaceptr++ = (FAST_FLOAT)(GETJSAMPLE(*elem_ptr++) - CENTERJSAMPLE);
              *workspaceptr++ = (FAST_FLOAT)(GETJSAMPLE(*elem_ptr++) - CENTERJSAMPLE);
              *workspaceptr++ = (FAST_FLOAT)(GETJSAMPLE(*elem_ptr++) - CENTERJSAMPLE);
              *workspaceptr++ = (FAST_FLOAT)(GETJSAMPLE(*elem_ptr++) - CENTERJSAMPLE);
#else
              { register int elem_c;
                for (elem_c = DCTSIZE; elem_c > 0; elem_c--) {
                    *workspaceptr++ = (FAST_FLOAT)
                        (GETJSAMPLE(*elem_ptr++) - CENTERJSAMPLE);
                }
              }
            }
        }
    }

    /* Perform the DCT */
    (*do_dct) (workspace);

    /* Quantize/descale the coefficients, and store into coef_blocks[] */
    { register FAST_FLOAT temp;
      register int i;
      register JCOEFPTR output_ptr = coef_blocks[bi];

```

```

        for (i = 0; i < DCTSIZE; i++) {
/* Apply the quantization and scaling factor */
temp = workspace[i] * divisors[i];
/* Round to nearest integer.
 * Since C does not specify the direction of rounding for negative
 * quotients, we have to force the dividend positive for portability.
 * The maximum coefficient size is +-16K (for 12-bit data), so this
 * code should work for either 16-bit or 32-bit ints.
 */
output_ptr[i] = (JCOEF) ((int) (temp + (FAST_FLOAT) 16384.5) - 16384);
        }
    }
}

#endif /* DCT_FLOAT_SUPPORTED */

/*
 * Initialize FDCT manager.
 */

GLOBAL(void)
jinit_forward_dct (j_compress_ptr cinfo)
{
    my_fdct_ptr fdct;
    int i;

    fdct = (my_fdct_ptr)
        (*cinfo->mem->alloc_small) ((j_common_ptr) cinfo, JPOOL_IMAGE,
                                   SIZEOF(my_fdct_controller));
    cinfo->fdct = (struct jpeg_forward_dct *) fdct;
    fdct->pub.start_pass = start_pass_fdctmgr;

    switch (cinfo->dct_method) {
#ifdef DCT_ISLOW_SUPPORTED
    case JDCT_ISLOW:
        fdct->pub.forward_DCT = forward_DCT;
        fdct->do_dct = jpeg_fdct_islow;
        break;
#endif
#ifdef DCT_IFAST_SUPPORTED
    case JDCT_IFAST:
        fdct->pub.forward_DCT = forward_DCT;
        fdct->do_dct = jpeg_fdct_ifast;
        break;
#endif
#ifdef DCT_FLOAT_SUPPORTED
    case JDCT_FLOAT:
        fdct->pub.forward_DCT = forward_DCT_float;
        fdct->do_float_dct = jpeg_fdct_float;
        break;
#endif
    default:
        ERREXIT(cinfo, JERR_NOT_COMPILED);
        break;
    }

    /* Mark divisor tables unallocated */
    for (i = 0; i < NUM_QUANT_TBLS; i++) {
        fdct->divisors[i] = NULL;
#ifdef DCT_FLOAT_SUPPORTED
        fdct->float_divisors[i] = NULL;
#endif
    }
}

```

```

/*
 * jchuff.c
 *
 * Copyright (C) 1991-1997, Thomas G. Lane.
 * This file is part of the Independent JPEG Group's software.
 * For conditions of distribution and use, see the accompanying README file.
 *
 * This file contains Huffman entropy encoding routines.
 *
 * Much of the complexity here has to do with supporting output suspension.
 * If the data destination module demands suspension, we want to be able to
 * back up to the start of the current MCU. To do this, we copy state
 * variables into local working storage, and update them back to the
 * permanent JPEG objects only upon successful completion of an MCU.
 */

#define JPEG_INTERNALS
#include "jinclude.h"
#include "jpeglib.h"
#include "jchuff.h" /* Declarations shared with jcphuff.c */

/* Expanded entropy encoder object for Huffman encoding.
 *
 * The savable_state subrecord contains fields that change within an MCU,
 * but must not be updated permanently until we complete the MCU.
 */

typedef struct {
    INT32 put_buffer; /* current bit-accumulation buffer */
    int put_bits; /* # of bits now in it */
    int last_dc_val[MAX_COMPS_IN_SCAN]; /* last DC coef for each component */
} savable_state;

/* This macro is to work around compilers with missing or broken
 * structure assignment. You'll need to fix this code if you have
 * such a compiler and you change MAX_COMPS_IN_SCAN.
 */
#ifdef NO_STRUCT_ASSIGN
#define ASSIGN_STATE(dest,src) ((dest) = (src))
#else
#define ASSIGN_STATE(dest,src) \
    if (MAX_COMPS_IN_SCAN == 4) \
        define ASSIGN_STATE(dest,src) \
        ((dest).put_buffer = (src).put_buffer, \
         (dest).put_bits = (src).put_bits, \
         (dest).last_dc_val[0] = (src).last_dc_val[0], \
         (dest).last_dc_val[1] = (src).last_dc_val[1], \
         (dest).last_dc_val[2] = (src).last_dc_val[2], \
         (dest).last_dc_val[3] = (src).last_dc_val[3]) \
    #endif
#endif

typedef struct {
    struct jpeg_entropy_encoder pub; /* public fields */

    savable_state saved; /* Bit buffer & DC state at start of MCU */

    /* These fields are NOT loaded into local working state. */
    unsigned int restarts_to_go; /* MCUs left in this restart interval */
    int next_restart_num; /* next restart number to write (0-7) */

    /* Pointers to derived tables (these workspaces have image lifespan) */
    c_derived_tbl * dc_derived_tbls[NUM_HUFF_TBLS];
    c_derived_tbl * ac_derived_tbls[NUM_HUFF_TBLS];

#ifdef ENTROPY_OPT_SUPPORTED /* Statistics tables for optimization */
    long * dc_count_ptrs[NUM_HUFF_TBLS];
    long * ac_count_ptrs[NUM_HUFF_TBLS];
#endif
} huff_entropy_encoder;

typedef huff_entropy_encoder * huff_entropy_ptr;

/* Working state while writing an MCU.
 * This struct contains all the fields that are needed by subroutines.
 */
typedef struct {

```

```

JOCTET * next_output_byte; /* => next byte to write in buffer
size_t free_in_buffer; /* of byte spaces remaining in buffer
savable_state cur; /* current bit buffer & DC state */
j_compress_ptr cinfo; /* dump_buffer needs access to this */
} working_state;

/* Forward declarations */
METHODDEF(boolean) encode_mcu_huff JPP((j_compress_ptr cinfo,
JBLOCKROW *MCU_data));
METHODDEF(void) finish_pass_huff JPP((j_compress_ptr cinfo));
#ifdef ENTROPY_OPT_SUPPORTED
METHODDEF(boolean) encode_mcu_gather JPP((j_compress_ptr cinfo,
JBLOCKROW *MCU_data));
METHODDEF(void) finish_pass_gather JPP((j_compress_ptr cinfo));
#endif

/*
 * Initialize for a Huffman-compressed scan.
 * If gather_statistics is TRUE, we do not output anything during the scan,
 * just count the Huffman symbols used and generate Huffman code tables.
 */

METHODDEF(void)
start_pass_huff (j_compress_ptr cinfo, boolean gather_statistics)
{
    huff_entropy_ptr entropy = (huff_entropy_ptr) cinfo->entropy;
    int ci, dctbl, actbl;
    jpeg_component_info * compptr;

    if (gather_statistics) {
#ifdef ENTROPY_OPT_SUPPORTED
        entropy->pub.encode_mcu = encode_mcu_gather;
        entropy->pub.finish_pass = finish_pass_gather;
    #else
        ERREXIT(cinfo, JERR_NOT_COMPILED);
    #endif
    } else {
        entropy->pub.encode_mcu = encode_mcu_huff;
        entropy->pub.finish_pass = finish_pass_huff;
    }

    for (ci = 0; ci < cinfo->comps_in_scan; ci++) {
        compptr = cinfo->cur_comp_info[ci];
        dctbl = compptr->dc_tbl_no;
        actbl = compptr->ac_tbl_no;
        if (gather_statistics) {
#ifdef ENTROPY_OPT_SUPPORTED
            /* Check for invalid table indexes */
            /* (make_c_derived_tbl does this in the other path) */
            if (dctbl < 0 || dctbl >= NUM_HUFF_TBLS)
                ERREXIT1(cinfo, JERR_NO_HUFF_TABLE, dctbl);
            if (actbl < 0 || actbl >= NUM_HUFF_TBLS)
                ERREXIT1(cinfo, JERR_NO_HUFF_TABLE, actbl);
            /* Allocate and zero the statistics tables */
            /* Note that jpeg_gen_optimal_table expects 257 entries in each table! */
            if (entropy->dc_count_ptrs[dctbl] == NULL)
                entropy->dc_count_ptrs[dctbl] = (long *)
                    (*cinfo->mem->alloc_small) ((j_common_ptr) cinfo, JPOOL_IMAGE,
                    257 * sizeof(long));
            MEMZERO(entropy->dc_count_ptrs[dctbl], 257 * sizeof(long));
            if (entropy->ac_count_ptrs[actbl] == NULL)
                entropy->ac_count_ptrs[actbl] = (long *)
                    (*cinfo->mem->alloc_small) ((j_common_ptr) cinfo, JPOOL_IMAGE,
                    257 * sizeof(long));
            MEMZERO(entropy->ac_count_ptrs[actbl], 257 * sizeof(long));
        #endif
        } else {
            /* Compute derived values for Huffman tables */
            /* We may do this more than once for a table, but it's not expensive */
            jpeg_make_c_derived_tbl(cinfo, TRUE, dctbl,
                & entropy->dc_derived_tbls[dctbl]);
            jpeg_make_c_derived_tbl(cinfo, FALSE, actbl,
                & entropy->ac_derived_tbls[actbl]);
        }
        /* Initialize DC predictions to 0 */
        entropy->saved_last_dc_val[ci] = 0;
    }
}

```

```

/* Initialize bit buffer to empty */
entropy->saved.put_buffer =
entropy->saved.put_bits = 0;

/* Initialize restart stuff */
entropy->restarts_to_go = cinfo->restart_interval;
entropy->next_restart_num = 0;
}

/*
 * Compute the derived values for a Huffman table.
 * This routine also performs some validation checks on the table.
 *
 * Note this is also used by jcphuff.c.
 */

GLOBAL(void)
jpeg_make_c_derived_tbl (j_compress_ptr cinfo, boolean isDC, int tblno,
                        c_derived_tbl ** pdtbl)
{
    JHUFF_TBL *htbl;
    c_derived_tbl *dtbl;
    int p, i, l, lastp, si, maxsymbol;
    char huffsize[257];
    unsigned int huffcode[257];
    unsigned int code;

    /* Note that huffsize[] and huffcode[] are filled in code-length order,
     * paralleling the order of the symbols themselves in htbl->huffval[].
     */

    /* Find the input Huffman table */
    if (tblno < 0 || tblno >= NUM_HUFF_TBLS)
        ERREXIT1(cinfo, JERR_NO_HUFF_TABLE, tblno);
    htbl =
        isDC ? cinfo->dc_huff_tbl_ptrs[tblno] : cinfo->ac_huff_tbl_ptrs[tblno];
    if (htbl == NULL)
        ERREXIT1(cinfo, JERR_NO_HUFF_TABLE, tblno);

    /* Allocate a workspace if we haven't already done so. */
    if (*pdtbl == NULL)
        *pdtbl = (c_derived_tbl *)
            (*cinfo->mem->alloc_small) ((j_common_ptr) cinfo, JPOOL_IMAGE,
                                       SIZEOF(c_derived_tbl));
    dtbl = *pdtbl;

    /* Figure C.1: make table of Huffman code length for each symbol */
    p = 0;
    for (l = 1; l <= 16; l++) {
        i = (int) htbl->bits[l];
        if (i < 0 || p + i > 256) /* protect against table overrun */
            ERREXIT(cinfo, JERR_BAD_HUFF_TABLE);
        while (i--)
            huffsize[p++] = (char) l;
    }
    huffsize[p] = 0;
    lastp = p;

    /* Figure C.2: generate the codes themselves */
    /* We also validate that the counts represent a legal Huffman code tree. */

    code = 0;
    si = huffsize[0];
    p = 0;
    while (huffsize[p]) {
        while (((int) huffsize[p]) == si) {
            huffcode[p++] = code;
            code++;
        }
        /* code is now 1 more than the last code used for codelength si; but
         * it must still fit in si bits, since no code is allowed to be all ones.
         */
        if (((INT32) code) >= (((INT32) 1) << si))
            ERREXIT(cinfo, JERR_BAD_HUFF_TABLE);
        code <<= 1;
        si++;
    }
}

```



```

/* Figure C.3: generate encoding tables */
/* These are code and size indexed by symbol value */

/* Set all codeless symbols to have code length 0;
 * this lets us detect duplicate VAL entries here, and later
 * allows emit_bits to detect any attempt to emit such symbols.
 */
MEMZERO(dtbl->ehufsi, SIZEOF(dtbl->ehufsi));

/* This is also a convenient place to check for out-of-range
 * and duplicated VAL entries. We allow 0..255 for AC symbols
 * but only 0..15 for DC. (We could constrain them further
 * based on data depth and mode, but this seems enough.)
 */
maxsymbol = isDC ? 15 : 255;

for (p = 0; p < lastp; p++) {
    i = htbl->huffval[p];
    if (i < 0 || i > maxsymbol || dtbl->ehufsi[i])
        ERREXIT(cinfo, JERR_BAD_HUFF_TABLE);
    dtbl->ehufco[i] = huffcode[p];
    dtbl->ehufsi[i] = huffsize[p];
}

/* Outputting bytes to the file */

/* Emit a byte, taking 'action' if must suspend. */
#define emit_byte(state, val, action) \
    ( (state->next_output_byte++ = (JOCTET) (val)); \
      if (--(state->free_in_buffer) == 0) \
          if (! dump_buffer(state)) \
              { action; } )

LOCAL(boolean)
dump_buffer (working_state * state)
/* Empty the output buffer; return TRUE if successful, FALSE if must suspend */
{
    struct jpeg_destination_mgr * dest = state->cinfo->dest;
    if (! (*dest->empty_output_buffer) (state->cinfo))
        return FALSE;
    /* After a successful buffer dump, must reset buffer pointers */
    state->next_output_byte = dest->next_output_byte;
    state->free_in_buffer = dest->free_in_buffer;
    return TRUE;
}

/* Outputting bits to the file */

/* Only the right 24 bits of put_buffer are used; the valid bits are
 * left-justified in this part. At most 16 bits can be passed to emit_bits
 * in one call, and we never retain more than 7 bits in put_buffer
 * between calls, so 24 bits are sufficient.
 */

INLINE
LOCAL(boolean)
emit_bits (working_state * state, unsigned int code, int size)
/* Emit some bits; return TRUE if successful, FALSE if must suspend */
{
    /* This routine is heavily used, so it's worth coding tightly. */
    register INT32 put_buffer = (INT32) code;
    register int put_bits = state->cur.put_bits;

    /* if size is 0, caller used an invalid Huffman table entry */
    if (size == 0)
        ERREXIT(state->cinfo, JERR_HUFF_MISSING_CODE);

    put_buffer &= (((INT32) 1) << size) - 1; /* mask off any extra bits in code */

    put_bits += size; /* new number of bits in buffer */

    put_buffer <<= 24 - put_bits; /* align incoming bits */

    put_buffer |= state->cur.put_buffer; /* and merge with old buffer contents */
}

```

```

while (put_bits >= 8) {
    int c = (int) ((put_buffer >> 16) & 0xFF);

    emit_byte(state, c, return FALSE);
    if (c == 0xFF) { /* need to stuff a zero byte? */
        emit_byte(state, 0, return FALSE);
    }
    put_buffer <<= 8;
    put_bits -= 8;
}

state->cur.put_buffer = put_buffer; /* update state variables */
state->cur.put_bits = put_bits;

return TRUE;
}

LOCAL(boolean)
flush_bits (working_state * state)
{
    if (! emit_bits(state, 0x7F, 7)) /* fill any partial byte with ones */
        return FALSE;
    state->cur.put_buffer = 0; /* and reset bit-buffer to empty */
    state->cur.put_bits = 0;
    return TRUE;
}

/* Encode a single block's worth of coefficients */

LOCAL(boolean)
encode_one_block (working_state * state, JCOEFPTR block, int last_dc_val,
                  c_derived_tbl *dctbl, c_derived_tbl *actbl)
{
    register int temp, temp2;
    register int nbits;
    register int k, r, i;

    /* Encode the DC coefficient difference per section F.1.2.1 */
    temp = temp2 = block[0] - last_dc_val;

    if (temp < 0) {
        temp = -temp; /* temp is abs value of input */
        /* For a negative input, want temp2 = bitwise complement of abs(input) */
        /* This code assumes we are on a two's complement machine */
        temp2--;
    }

    /* Find the number of bits needed for the magnitude of the coefficient */
    nbits = 0;
    while (temp) {
        nbits++;
        temp >>= 1;
    }
    /* Check for out-of-range coefficient values.
     * Since we're encoding a difference, the range limit is twice as much.
     */
    if (nbits > MAX_COEF_BITS+1)
        ERREXIT(state->cinfo, JERR_BAD_DCT_COEF);

    /* Emit the Huffman-coded symbol for the number of bits */
    if (! emit_bits(state, dctbl->ehufco[nbits], dctbl->ehufsi[nbits]))
        return FALSE;

    /* Emit that number of bits of the value, if positive, */
    /* or the complement of its magnitude, if negative. */
    if (nbits) /* emit_bits rejects calls with size 0 */
        if (! emit_bits(state, (unsigned int) temp2, nbits))
            return FALSE;

    /* Encode the AC coefficients per section F.1.2.2 */

    r = 0; /* r = run length of zeros */

    for (k = 1; k < DCTSIZE2; k++) {
        if ((temp = block[jpeg_natural_order[k]]) == 0) {
            r++;
        } else {

```

```

    /* if run length > 15, emit special run-length-16 codes */
    while (r > 15) {
        if (! emit_bits(state, actbl->ehufco[0xF0], actbl->ehufsi[0xF0]))
            return FALSE;
        r -= 16;
    }

    temp2 = temp;
    if (temp < 0) {
        temp = -temp; /* temp is abs value of input */
        /* This code assumes we are on a two's complement machine */
        temp2--;
    }

    /* Find the number of bits needed for the magnitude of the coefficient */
    nbits = 1; /* there must be at least one 1 bit */
    while ((temp >= 1))
        nbits++;
    /* Check for out-of-range coefficient values */
    if (nbits > MAX_COEF_BITS)
        ERREXIT(state->cinfo, JERR_BAD_DCT_COEF);

    /* Emit Huffman symbol for run length / number of bits */
    i = (r << 4) + nbits;
    if (! emit_bits(state, actbl->ehufco[i], actbl->ehufsi[i]))
        return FALSE;

    /* Emit that number of bits of the value, if positive, */
    /* or the complement of its magnitude, if negative. */
    if (! emit_bits(state, (unsigned int) temp2, nbits))
        return FALSE;

    r = 0;
}

/* If the last coef(s) were zero, emit an end-of-block code */
if (r > 0)
    if (! emit_bits(state, actbl->ehufco[0], actbl->ehufsi[0]))
        return FALSE;
return TRUE;
}

/* Emit a restart marker & resynchronize predictions.
LOCAL(boolean)
emit_restart (working_state * state, int restart_num)
{
    int ci;

    if (! flush_bits(state))
        return FALSE;

    emit_byte(state, 0xFF, return FALSE);
    emit_byte(state, JPEG_RST0 + restart_num, return FALSE);

    /* Re-initialize DC predictions to 0 */
    for (ci = 0; ci < state->cinfo->comps_in_scan; ci++)
        state->cur.last_dc_val[ci] = 0;

    /* The restart counter is not updated until we successfully write the MCU. */

    return TRUE;
}

/*
 * Encode and output one MCU's worth of Huffman-compressed coefficients.
 */

METHODDEF(boolean)
encode_mcu_huff (j_compress_ptr cinfo, JBLOCKROW *MCU_data)
{
    huff_entropy_ptr entropy = (huff_entropy_ptr) cinfo->entropy;
    working_state state;
    int blkn, ci;

```

```

jpeg_component_info * comp_ptr;

/* Load up working state */
state.next_output_byte = cinfo->dest->next_output_byte;
state.free_in_buffer = cinfo->dest->free_in_buffer;
ASSIGN_STATE(state.cur, entropy->saved);
state.cinfo = cinfo;

/* Emit restart marker if needed */
if (cinfo->restart_interval) {
    if (entropy->restarts_to_go == 0)
        if (! emit_restart(&state, entropy->next_restart_num))
            return FALSE;
}

/* Encode the MCU data blocks */
for (blk_n = 0; blk_n < cinfo->blocks_in_MCU; blk_n++) {
    ci = cinfo->MCU_membership[blk_n];
    comp_ptr = cinfo->cur_comp_info[ci];
    if (! encode_one_block(&state,
        MCU_data[blk_n][0], state.cur.last_dc_val[ci],
        entropy->dc_derived_tbls[comp_ptr->dc_tbl_no],
        entropy->ac_derived_tbls[comp_ptr->ac_tbl_no]))
        return FALSE;
    /* Update last_dc_val */
    state.cur.last_dc_val[ci] = MCU_data[blk_n][0][0];
}

/* Completed MCU, so update state */
cinfo->dest->next_output_byte = state.next_output_byte;
cinfo->dest->free_in_buffer = state.free_in_buffer;
ASSIGN_STATE(entropy->saved, state.cur);

/* Update restart-interval state too */
if (cinfo->restart_interval) {
    if (entropy->restarts_to_go == 0) {
        entropy->restarts_to_go = cinfo->restart_interval;
        entropy->next_restart_num++;
        entropy->next_restart_num &= 7;
    }
    entropy->restarts_to_go--;
}

return TRUE;
}

/* Finish up at the end of a Huffman-compressed scan.
METHODDEF(void)
finish_pass_huff (j_compress_ptr cinfo)
{
    huff_entropy_ptr entropy = (huff_entropy_ptr) cinfo->entropy;
    working_state state;

    /* Load up working state ... flush_bits needs it */
    state.next_output_byte = cinfo->dest->next_output_byte;
    state.free_in_buffer = cinfo->dest->free_in_buffer;
    ASSIGN_STATE(state.cur, entropy->saved);
    state.cinfo = cinfo;

    /* Flush out the last data */
    if (! flush_bits(&state))
        ERREXIT(cinfo, JERR_CANT_SUSPEND);

    /* Update state */
    cinfo->dest->next_output_byte = state.next_output_byte;
    cinfo->dest->free_in_buffer = state.free_in_buffer;
    ASSIGN_STATE(entropy->saved, state.cur);
}

/*
 * Huffman coding optimization.
 *
 * We first scan the supplied data and count the number of uses of each symbol
 * that is to be Huffman-coded. (This process MUST agree with the code above.)
 * Then we build a Huffman coding tree for the observed counts.

```

```

* Symbols which are not needed at all for the particular image are not
* assigned any code, which saves space in the DHT marker as well as in
* the compressed data.
*/

```

```

#ifdef ENTROPY_OPT_SUPPORTED

```

```

/* Process a single block's worth of coefficients */

```

```

LOCAL(void)

```

```

htest_one_block (j_compress_ptr cinfo, JCOEFPTR block, int last_dc_val,
                 long dc_counts[], long ac_counts[])

```

```

{
    register int temp;
    register int nbits;
    register int k, r;

    /* Encode the DC coefficient difference per section F.1.2.1 */

    temp = block[0] - last_dc_val;
    if (temp < 0)
        temp = -temp;

    /* Find the number of bits needed for the magnitude of the coefficient */
    nbits = 0;
    while (temp) {
        nbits++;
        temp >>= 1;
    }
    /* Check for out-of-range coefficient values.
     * Since we're encoding a difference, the range limit is twice as much.
     */
    if (nbits > MAX_COEF_BITS+1)
        ERREXIT(cinfo, JERR_BAD_DCT_COEF);

    /* Count the Huffman symbol for the number of bits */
    dc_counts[nbits]++;

    /* Encode the AC coefficients per section F.1.2.2 */
    r = 0;          /* r = run length of zeros */
    for (k = 1; k < DCTSIZE2; k++) {
        if ((temp = block[jpeg_natural_order[k]]) == 0) {
            r++;
        } else {
            /* if run length > 15, must emit special run-length-16 codes (0xF0) */
            while (r > 15) {
                ac_counts[0xF0]++;
                r -= 16;
            }

            /* Find the number of bits needed for the magnitude of the coefficient */
            if (temp < 0)
                temp = -temp;

            /* Find the number of bits needed for the magnitude of the coefficient */
            nbits = 1;          /* there must be at least one 1 bit */
            while ((temp >>= 1))
                nbits++;
            /* Check for out-of-range coefficient values */
            if (nbits > MAX_COEF_BITS)
                ERREXIT(cinfo, JERR_BAD_DCT_COEF);

            /* Count Huffman symbol for run length / number of bits */
            ac_counts[(r < 4) + nbits]++;

            r = 0;
        }
    }

    /* If the last coef(s) were zero, emit an end-of-block code */
    if (r > 0)
        ac_counts[0]++;
}

/*
 * Trial-encode one MCU's worth of Huffman-compressed coefficients.
 */

```

```

* No data is actually output, so no suspension return is possible
*/

```

```

METHODDEF(boolean)
encode_mcu_gather (j_compress_ptr cinfo, JBLOCKROW *MCU_data)
{
    huff_entropy_ptr entropy = (huff_entropy_ptr) cinfo->entropy;
    int blkcn, ci;
    jpeg_component_info * compptr;

    /* Take care of restart intervals if needed */
    if (cinfo->restart_interval) {
        if (entropy->restarts_to_go == 0) {
            /* Re-initialize DC predictions to 0 */
            for (ci = 0; ci < cinfo->comps_in_scan; ci++)
                entropy->saved.last_dc_val[ci] = 0;
            /* Update restart state */
            entropy->restarts_to_go = cinfo->restart_interval;
        }
        entropy->restarts_to_go--;
    }

    for (blkcn = 0; blkcn < cinfo->blocks_in_MCU; blkcn++) {
        ci = cinfo->MCU_membership[blkcn];
        compptr = cinfo->cur_comp_info[ci];
        htest_one_block(cinfo, MCU_data[blkcn][0], entropy->saved.last_dc_val[ci],
            entropy->dc_count_ptrs[compptr->dc_tbl_no],
            entropy->ac_count_ptrs[compptr->ac_tbl_no]);
        entropy->saved.last_dc_val[ci] = MCU_data[blkcn][0][0];
    }
}

```

```

    return TRUE;
}

```

```

/* Generate the best Huffman code table for the given counts, fill htbl.
Note this is also used by jcp Huff.c.
*/

```

```

/* The JPEG standard requires that no symbol be assigned a codeword of all
one bits (so that padding bits added at the end of a compressed segment
can't look like a valid code). Because of the canonical ordering of
codewords, this just means that there must be an unused slot in the
longest codeword length category. Section K.2 of the JPEG spec suggests
reserving such a slot by pretending that symbol 256 is a valid symbol
with count 1. In theory that's not optimal; giving it count zero but
including it in the symbol set anyway should give a better Huffman code.
But the theoretically better code actually seems to come out worse in
practice, because it produces more all-ones bytes (which incur stuffed
zero bytes in the final file). In any case the difference is tiny.
*/

```

```

/* The JPEG standard requires Huffman codes to be no more than 16 bits long.
If some symbols have a very small but nonzero probability, the Huffman tree
must be adjusted to meet the code length restriction. We currently use
the adjustment method suggested in JPEG section K.2. This method is *not*
optimal; it may not choose the best possible limited-length code. But
typically only very-low-frequency symbols will be given less-than-optimal
lengths, so the code is almost optimal. Experimental comparisons against
an optimal limited-length-code algorithm indicate that the difference is
microscopic --- usually less than a hundredth of a percent of total size.
So the extra complexity of an optimal algorithm doesn't seem worthwhile.
*/

```

```

GLOBAL(void)
jpeg_gen_optimal_table (j_compress_ptr cinfo, JHUFF_TBL * htbl, long freq[])
{
#define MAX_CLEN 32 /* assumed maximum initial code length */
    UINT8 bits[MAX_CLEN+1]; /* bits[k] = # of symbols with code length k */
    int codesize[257]; /* codesize[k] = code length of symbol k */
    int others[257]; /* next symbol in current branch of tree */
    int c1, c2;
    int p, i, j;
    long v;

    /* This algorithm is explained in section K.2 of the JPEG standard */

    MEMZERO(bits, sizeof(bits));
    MEMZERO(codesize, sizeof(codesize));
    for (i = 0; i < 257; i++)
        others[i] = -1; /* init links to empty */
}

```

```

freq[256] = 1;          /* make sure 256 has a nonzero count */
/* Including the pseudo-symbol 256 in the Huffman procedure guarantees
 * that no real symbol is given code-value of all ones, because 256
 * will be placed last in the largest codeword category.
 */

/* Huffman's basic algorithm to assign optimal code lengths to symbols */

for (;;) {
    /* Find the smallest nonzero frequency, set c1 = its symbol */
    /* In case of ties, take the larger symbol number */
    c1 = -1;
    v = 1000000000L;
    for (i = 0; i <= 256; i++) {
        if (freq[i] && freq[i] <= v) {
            v = freq[i];
            c1 = i;
        }
    }

    /* Find the next smallest nonzero frequency, set c2 = its symbol */
    /* In case of ties, take the larger symbol number */
    c2 = -1;
    v = 1000000000L;
    for (i = 0; i <= 256; i++) {
        if (freq[i] && freq[i] <= v && i != c1) {
            v = freq[i];
            c2 = i;
        }
    }

    /* Done if we've merged everything into one frequency */
    if (c2 < 0)
        break;

    /* Else merge the two counts/trees */
    freq[c1] += freq[c2];
    freq[c2] = 0;

    /* Increment the codesize of everything in c1's tree branch */
    codesize[c1]++;
    while (others[c1] >= 0) {
        c1 = others[c1];
        codesize[c1]++;
    }

    others[c1] = c2;          /* chain c2 onto c1's tree branch */

    /* Increment the codesize of everything in c2's tree branch */
    codesize[c2]++;
    while (others[c2] >= 0) {
        c2 = others[c2];
        codesize[c2]++;
    }
}

/* Now count the number of symbols of each code length */
for (i = 0; i <= 256; i++) {
    if (codesize[i]) {
        /* The JPEG standard seems to think that this can't happen, */
        /* but I'm paranoid... */
        if (codesize[i] > MAX_CLEN)
            ERREXIT(cinfo, JERR_HUFF_CLEN_OVERFLOW);

        bits[codesize[i]]++;
    }
}

/* JPEG doesn't allow symbols with code lengths over 16 bits, so if the pure
 * Huffman procedure assigned any such lengths, we must adjust the coding.
 * Here is what the JPEG spec says about how this next bit works:
 * Since symbols are paired for the longest Huffman code, the symbols are
 * removed from this length category two at a time. The prefix for the pair
 * (which is one bit shorter) is allocated to one of the pair; then,
 * skipping the BITS entry for that prefix length, a code word from the next
 * shortest nonzero BITS entry is converted into a prefix for two code words
 * one bit longer.
 */

```

```

for (i = MAX_CLEN; i > 16; i--) {
    while (bits[i] > 0) {
        j = i - 2; /* find length of new prefix to be used */
        while (bits[j] == 0)
            j--;

        bits[i] -= 2; /* remove two symbols */
        bits[i-1]++; /* one goes in this length */
        bits[j+1] += 2; /* two new symbols in this length */
        bits[j]--; /* symbol of this length is now a prefix */
    }
}

/* Remove the count for the pseudo-symbol 256 from the largest codelength */
while (bits[i] == 0) /* find largest codelength still in use */
    i--;
bits[i]--;

/* Return final symbol counts (only for lengths 0..16) */
MEMCOPY(htbl->bits, bits, SIZEOF(htbl->bits));

/* Return a list of the symbols sorted by code length */
/* It's not real clear to me why we don't need to consider the codelength
 * changes made above, but the JPEG spec seems to think this works.
 */
p = 0;
for (i = 1; i <= MAX_CLEN; i++) {
    for (j = 0; j <= 255; j++) {
        if (codesize[j] == i) {
            htbl->huffval[p] = (UINT8) j;
            p++;
        }
    }
}

/* Set sent_table FALSE so updated table will be written to JPEG file. */
htbl->sent_table = FALSE;
}

/* Finish up a statistics-gathering pass and create the new Huffman tables.
 */
METHODDEF(void)
finish_pass_gather (j_compress_ptr cinfo)
{
    huff_entropy_ptr entropy = (huff_entropy_ptr) cinfo->entropy;
    int ci, dctbl, actbl;
    jpeg_component_info * compptr;
    JHUFF_TBL **htblptr;
    boolean did_dc[NUM_HUFF_TBLS];
    boolean did_ac[NUM_HUFF_TBLS];

    /* It's important not to apply jpeg_gen_optimal_table more than once
     * per table, because it clobbers the input frequency counts!
     */
    MEMZERO(did_dc, SIZEOF(did_dc));
    MEMZERO(did_ac, SIZEOF(did_ac));

    for (ci = 0; ci < cinfo->comps_in_scan; ci++) {
        compptr = cinfo->cur_comp_info[ci];
        dctbl = compptr->dc_tbl_no;
        actbl = compptr->ac_tbl_no;
        if (! did_dc[dctbl]) {
            htblptr = & cinfo->dc_huff_tbl_ptrs[dctbl];
            if (*htblptr == NULL)
                *htblptr = jpeg_alloc_huff_table((j_common_ptr) cinfo);
            jpeg_gen_optimal_table(cinfo, *htblptr, entropy->dc_count_ptrs[dctbl]);
            did_dc[dctbl] = TRUE;
        }
        if (! did_ac[actbl]) {
            htblptr = & cinfo->ac_huff_tbl_ptrs[actbl];
            if (*htblptr == NULL)
                *htblptr = jpeg_alloc_huff_table((j_common_ptr) cinfo);
            jpeg_gen_optimal_table(cinfo, *htblptr, entropy->ac_count_ptrs[actbl]);
            did_ac[actbl] = TRUE;
        }
    }
}
}

```





```

/*
 * jcinit.c
 *
 * Copyright (C) 1991-1997, Thomas G. Lane.
 * This file is part of the Independent JPEG Group's software.
 * For conditions of distribution and use, see the accompanying README file.
 *
 * This file contains initialization logic for the JPEG compressor.
 * This routine is in charge of selecting the modules to be executed and
 * making an initialization call to each one.
 *
 * Logically, this code belongs in jcmaster.c. It's split out because
 * linking this routine implies linking the entire compression library.
 * For a transcoding-only application, we want to be able to use jcmaster.c
 * without linking in the whole library.
 */

#define JPEG_INTERNALS
#include "jinclude.h"
#include "jpeglib.h"

/*
 * Master selection of compression modules.
 * This is done once at the start of processing an image. We determine
 * which modules will be used and give them appropriate initialization calls.
 */

GLOBAL(void)
jinit_compress_master (j_compress_ptr cinfo)
{
  /* Initialize master control (includes parameter checking/processing) */
  jinit_c_master_control(cinfo, FALSE /* full compression */);

  /* Preprocessing */
  if (! cinfo->raw_data_in) {
    jinit_color_converter(cinfo);
    jinit_downsampler(cinfo);
    jinit_c_prep_controller(cinfo, FALSE /* never need full buffer here */);
  }

  /* Forward DCT */
  jinit_forward_dct(cinfo);

  /* Entropy encoding: either Huffman or arithmetic coding. */
  if (cinfo->arith_code) {
    ERREXIT(cinfo, JERR_ARITH_NOTIMPL);
  } else {
    if (cinfo->progressive_mode) {
#ifdef C_PROGRESSIVE_SUPPORTED
      jinit_phuff_encoder(cinfo);
    #else
      ERREXIT(cinfo, JERR_NOT_COMPILED);
    #endif
    } else
      jinit_huff_encoder(cinfo);
  }

  /* Need a full-image coefficient buffer in any multi-pass mode. */
  jinit_c_coef_controller(cinfo,
    (boolean) (cinfo->num_scans > 1 || cinfo->optimize_coding));
  jinit_c_main_controller(cinfo, FALSE /* never need full buffer here */);

  jinit_marker_writer(cinfo);

  /* We can now tell the memory manager to allocate virtual arrays. */
  (*cinfo->mem->realize_virt_arrays) ((j_common_ptr) cinfo);

  /* Write the datastream header (SOI) immediately.
   * Frame and scan headers are postponed till later.
   * This lets application insert special markers after the SOI.
   */
  (*cinfo->marker->write_file_header) (cinfo);
}

```

```

/*
 * jcmaint.c
 *
 * Copyright (C) 1994-1996, Thomas G. Lane.
 * This file is part of the Independent JPEG Group's software.
 * For conditions of distribution and use, see the accompanying README file.
 *
 * This file contains the main buffer controller for compression.
 * The main buffer lies between the pre-processor and the JPEG
 * compressor proper; it holds downsampled data in the JPEG colorspace.
 */

#define JPEG_INTERNALS
#include "jinclude.h"
#include "jpeglib.h"

/* Note: currently, there is no operating mode in which a full-image buffer
 * is needed at this step.  If there were, that mode could not be used with
 * "raw data" input, since this module is bypassed in that case.  However,
 * we've left the code here for possible use in special applications.
 */
#undef FULL_MAIN_BUFFER_SUPPORTED

/* Private buffer controller object */
typedef struct {
  struct jpeg_c_main_controller pub; /* public fields */

  JDIMENSION cur_iMCU_row; /* number of current iMCU row */
  JDIMENSION rowgroup_ctr; /* counts row groups received in iMCU row */
  boolean suspended; /* remember if we suspended output */
  J_BUF_MODE pass_mode; /* current operating mode */

  /* If using just a strip buffer, this points to the entire set of buffers
   * (we allocate one for each component).  In the full-image case, this
   * points to the currently accessible strips of the virtual arrays.
   */
  JSAMPARRAY buffer[MAX_COMPONENTS];

#ifdef FULL_MAIN_BUFFER_SUPPORTED
  /* If using full-image storage, this array holds pointers to virtual-array
   * control blocks for each component.  Unused if not full-image storage.
   */
  jvirt_sarray_ptr whole_image[MAX_COMPONENTS];
#endif
} my_main_controller;

typedef my_main_controller * my_main_ptr;

/* Forward declarations */
METHODDEF(void) process_data_simple_main
  (JPP((j_compress_ptr cinfo, JSAMPARRAY input_buf,
        JDIMENSION *in_row_ctr, JDIMENSION in_rows_avail)));
#ifdef FULL_MAIN_BUFFER_SUPPORTED
METHODDEF(void) process_data_buffer_main
  (JPP((j_compress_ptr cinfo, JSAMPARRAY input_buf,
        JDIMENSION *in_row_ctr, JDIMENSION in_rows_avail)));
#endif

/*
 * Initialize for a processing pass.
 */
METHODDEF(void)
start_pass_main (j_compress_ptr cinfo, J_BUF_MODE pass_mode)
{
  my_main_ptr main = (my_main_ptr) cinfo->main;

  /* Do nothing in raw-data mode. */
  if (cinfo->raw_data_in)
    return;

  main->cur_iMCU_row = 0; /* initialize counters */
  main->rowgroup_ctr = 0;
  main->suspended = FALSE;
  main->pass_mode = pass_mode; /* save mode for use by process_data */
}

```

```

switch (pass_mode) {
case JBUF_PASS_THRU:
#ifdef FULL_MAIN_BUFFER_SUPPORTED
    if (main->whole_image[0] != NULL)
        ERREXIT(cinfo, JERR_BAD_BUFFER_MODE);
#endif
    main->pub.process_data = process_data_simple_main;
    break;
#ifdef FULL_MAIN_BUFFER_SUPPORTED
case JBUF_SAVE_SOURCE:
case JBUF_CRANK_DEST:
case JBUF_SAVE_AND_PASS:
    if (main->whole_image[0] == NULL)
        ERREXIT(cinfo, JERR_BAD_BUFFER_MODE);
    main->pub.process_data = process_data_buffer_main;
    break;
#endif
default:
    ERREXIT(cinfo, JERR_BAD_BUFFER_MODE);
    break;
}
}

/*
 * Process some data.
 * This routine handles the simple pass-through mode,
 * where we have only a strip buffer.
 */

METHODDEF(void)
process_data_simple_main (j_compress_ptr cinfo,
                          JSAMPARRAY input_buf, JDIMENSION *in_row_ctr,
                          JDIMENSION in_rows_avail)
{
    my_main_ptr main = (my_main_ptr) cinfo->main;

    while (main->cur_iMCU_row < cinfo->total_iMCU_rows) {
        /* Read input data if we haven't filled the main buffer yet */
        if (main->rowgroup_ctr < DCTSIZE)
            (*cinfo->prep->pre_process_data) (cinfo,
                                              input_buf, in_row_ctr, in_rows_avail,
                                              main->buffer, &main->rowgroup_ctr,
                                              (JDIMENSION) DCTSIZE);

        /* If we don't have a full iMCU row buffered, return to application for
         * more data. Note that preprocessor will always pad to fill the iMCU row
         * at the bottom of the image.
         */
        if (main->rowgroup_ctr != DCTSIZE)
            return;

        /* Send the completed row to the compressor */
        if (! (*cinfo->coef->compress_data) (cinfo, main->buffer)) {
            /* If compressor did not consume the whole row, then we must need to
             * suspend processing and return to the application. In this situation
             * we pretend we didn't yet consume the last input row; otherwise, if
             * it happened to be the last row of the image, the application would
             * think we were done.
             */
            if (! main->suspended) {
                (*in_row_ctr)--;
                main->suspended = TRUE;
            }
            return;
        }
        /* We did finish the row. Undo our little suspension hack if a previous
         * call suspended; then mark the main buffer empty.
         */
        if (main->suspended) {
            (*in_row_ctr)++;
            main->suspended = FALSE;
        }
        main->rowgroup_ctr = 0;
        main->cur_iMCU_row++;
    }
}

```

```
#ifdef FULL_MAIN_BUFFER_SUPPORTED
```

```
/*
 * Process some data.
 * This routine handles all of the modes that use a full-size buffer.
 */
```

```
METHODDEF(void)
```

```
process_data_buffer_main (j_compress_ptr cinfo,
                          JSAMPARRAY input_buf, JDIMENSION *in_row_ctr,
                          JDIMENSION in_rows_avail)
```

```
{
    my_main_ptr main = (my_main_ptr) cinfo->main;
    int ci;
    jpeg_component_info *comp_ptr;
    boolean writing = (main->pass_mode != JBUF_CRANK_DEST);

    while (main->cur_iMCU_row < cinfo->total_iMCU_rows) {
        /* Realign the virtual buffers if at the start of an iMCU row. */
        if (main->rowgroup_ctr == 0) {
            for (ci = 0, comp_ptr = cinfo->comp_info; ci < cinfo->num_components;
                 ci++, comp_ptr++) {
                main->buffer[ci] = (*cinfo->mem->access_virt_sarray)
                    ((j_common_ptr) cinfo, main->whole_image[ci],
                     main->cur_iMCU_row * (comp_ptr->v_samp_factor * DCTSIZE),
                     (JDIMENSION) (comp_ptr->v_samp_factor * DCTSIZE), writing);
            }
            /* In a read pass, pretend we just read some source data. */
            if (! writing) {
                *in_row_ctr += cinfo->max_v_samp_factor * DCTSIZE;
                main->rowgroup_ctr = DCTSIZE;
            }
        }

        /* If a write pass, read input data until the current iMCU row is full. */
        /* Note: preprocessor will pad if necessary to fill the last iMCU row. */
        if (writing) {
            (*cinfo->prep->pre_process_data) (cinfo,
                                              input_buf, in_row_ctr, in_rows_avail,
                                              main->buffer, &main->rowgroup_ctr,
                                              (JDIMENSION) DCTSIZE);
            /* Return to application if we need more data to fill the iMCU row. */
            if (main->rowgroup_ctr < DCTSIZE)
                return;
        }

        /* Emit data, unless this is a sink-only pass. */
        if (main->pass_mode != JBUF_SAVE_SOURCE) {
            if (! (*cinfo->coef->compress_data) (cinfo, main->buffer)) {
                /* If compressor did not consume the whole row, then we must need to
                 * suspend processing and return to the application. In this situation
                 * we pretend we didn't yet consume the last input row; otherwise, if
                 * it happened to be the last row of the image, the application would
                 * think we were done.
                 */
                if (! main->suspended) {
                    (*in_row_ctr)--;
                    main->suspended = TRUE;
                }
                return;
            }
            /* We did finish the row. Undo our little suspension hack if a previous
             * call suspended; then mark the main buffer empty.
             */
            if (main->suspended) {
                (*in_row_ctr)++;
                main->suspended = FALSE;
            }
        }

        /* If get here, we are done with this iMCU row. Mark buffer empty. */
        main->rowgroup_ctr = 0;
        main->cur_iMCU_row++;
    }
}
```

```
#endif /* FULL_MAIN_BUFFER_SUPPORTED */
```

```
/*
```

```

* Initialize main buffer controller.
*/

GLOBAL(void)
jinit_c_main_controller (j_compress_ptr cinfo, boolean need_full_buffer)
{
    my_main_ptr main;
    int ci;
    jpeg_component_info *comp_ptr;

    main = (my_main_ptr)
        (*cinfo->mem->alloc_small) ((j_common_ptr) cinfo, JPOOL_IMAGE,
            SIZEOF(my_main_controller));
    cinfo->main = (struct jpeg_c_main_controller *) main;
    main->pub.start_pass = start_pass_main;

    /* We don't need to create a buffer in raw-data mode. */
    if (cinfo->raw_data_in)
        return;

    /* Create the buffer. It holds downsampled data, so each component
     * may be of a different size.
     */
    if (need_full_buffer) {
#ifdef FULL_MAIN_BUFFER_SUPPORTED
        /* Allocate a full-image virtual array for each component */
        /* Note we pad the bottom to a multiple of the iMCU height */
        for (ci = 0, comp_ptr = cinfo->comp_info; ci < cinfo->num_components;
            ci++, comp_ptr++) {
            main->whole_image[ci] = (*cinfo->mem->request_virt_sarray)
                ((j_common_ptr) cinfo, JPOOL_IMAGE, FALSE,
                comp_ptr->width_in_blocks * DCTSIZE,
                (JDIMENSION) jround_up((long) comp_ptr->height_in_blocks,
                    (long) comp_ptr->v_samp_factor) * DCTSIZE,
                (JDIMENSION) (comp_ptr->v_samp_factor * DCTSIZE));
        }
#else
        ERREXIT(cinfo, JERR_BAD_BUFFER_MODE);
#endif
    } else {
#ifdef FULL_MAIN_BUFFER_SUPPORTED
        main->whole_image[0] = NULL; /* flag for no virtual arrays */
#endif
        /* Allocate a strip buffer for each component */
        for (ci = 0, comp_ptr = cinfo->comp_info; ci < cinfo->num_components;
            ci++, comp_ptr++) {
            main->buffer[ci] = (*cinfo->mem->alloc_sarray)
                ((j_common_ptr) cinfo, JPOOL_IMAGE,
                comp_ptr->width_in_blocks * DCTSIZE,
                (JDIMENSION) (comp_ptr->v_samp_factor * DCTSIZE));
        }
    }
}

```

```

/*
 * jcmarker.c
 *
 * Copyright (C) 1991-1998, Thomas G. Lane.
 * This file is part of the Independent JPEG Group's software.
 * For conditions of distribution and use, see the accompanying README file.
 *
 * This file contains routines to write JPEG datastream markers.
 */

```

```

#define JPEG_INTERNALS
#include "jinclude.h"
#include "jpeglib.h"

```

```

typedef enum {          /* JPEG marker codes */

```

```

    M_SOF0  = 0xc0,
    M_SOF1  = 0xc1,
    M_SOF2  = 0xc2,
    M_SOF3  = 0xc3,

```

```

    M_SOF5  = 0xc5,
    M_SOF6  = 0xc6,
    M_SOF7  = 0xc7,

```

```

    M_JPG   = 0xc8,
    M_SOF9  = 0xc9,
    M_SOF10 = 0xca,
    M_SOF11 = 0xcb,

```

```

    M_SOF13 = 0xcd,
    M_SOF14 = 0xce,
    M_SOF15 = 0xcf,

```

```

    M_DHT   = 0xcc,

```

```

    M_DAC   = 0xcc,

```

```

    M_RST0  = 0xd0,

```

```

    M_RST1  = 0xd1,

```

```

    M_RST2  = 0xd2,

```

```

    M_RST3  = 0xd3,

```

```

    M_RST4  = 0xd4,

```

```

    M_RST5  = 0xd5,

```

```

    M_RST6  = 0xd6,

```

```

    M_RST7  = 0xd7,

```

```

    M_SOI   = 0xd8,

```

```

    M_EOI   = 0xd9,

```

```

    M_SOS   = 0xda,

```

```

    M_DQT   = 0xdb,

```

```

    M_DNL   = 0xdc,

```

```

    M_DRI   = 0xdd,

```

```

    M_DHP   = 0xde,

```

```

    M_EXP   = 0xdf,

```

```

    M_APP0  = 0xe0,

```

```

    M_APP1  = 0xe1,

```

```

    M_APP2  = 0xe2,

```

```

    M_APP3  = 0xe3,

```

```

    M_APP4  = 0xe4,

```

```

    M_APP5  = 0xe5,

```

```

    M_APP6  = 0xe6,

```

```

    M_APP7  = 0xe7,

```

```

    M_APP8  = 0xe8,

```

```

    M_APP9  = 0xe9,

```

```

    M_APP10 = 0xea,

```

```

    M_APP11 = 0xeb,

```

```

    M_APP12 = 0xec,

```

```

    M_APP13 = 0xed,

```

```

    M_APP14 = 0xee,

```

```

    M_APP15 = 0xef,

```

```

    M_JPG0  = 0xf0,

```

```

    M_JPG13 = 0xfd,

```

```

    M_COM   = 0xfe,

```

```

    M_TEM   = 0x01,

```

```

    M_ERROR = 0x100

```

```

} JPEG_MARKER;

/* Private state */

typedef struct {
    struct jpeg_marker_writer pub; /* public fields */

    unsigned int last_restart_interval; /* last DRI value emitted; 0 after SOI */
} my_marker_writer;

typedef my_marker_writer * my_marker_ptr;

/*
 * Basic output routines.
 *
 * Note that we do not support suspension while writing a marker.
 * Therefore, an application using suspension must ensure that there is
 * enough buffer space for the initial markers (typ. 600-700 bytes) before
 * calling jpeg_start_compress, and enough space to write the trailing EOI
 * (a few bytes) before calling jpeg_finish_compress. Multipass compression
 * modes are not supported at all with suspension, so those two are the only
 * points where markers will be written.
 */

LOCAL(void)
emit_byte (j_compress_ptr cinfo, int val)
/* Emit a byte */
{
    struct jpeg_destination_mgr * dest = cinfo->dest;

    *(dest->next_output_byte)++ = (JOCTET) val;
    if (--dest->free_in_buffer == 0) {
        if (! (*dest->empty_output_buffer) (cinfo))
            ERREXIT(cinfo, JERR_CANT_SUSPEND);
    }
}

LOCAL(void)
emit_marker (j_compress_ptr cinfo, JPEG_MARKER mark)
/* Emit a marker code */
{
    emit_byte(cinfo, 0xFF);
    emit_byte(cinfo, (int) mark);
}

LOCAL(void)
emit_2bytes (j_compress_ptr cinfo, int value)
/* Emit a 2-byte integer; these are always MSB first in JPEG files */
{
    emit_byte(cinfo, (value >> 8) & 0xFF);
    emit_byte(cinfo, value & 0xFF);
}

/*
 * Routines to write specific marker types.
 */

LOCAL(int)
emit_dqt (j_compress_ptr cinfo, int index)
/* Emit a DQT marker */
/* Returns the precision used (0 = 8bits, 1 = 16bits) for baseline checking */
{
    JQUANT_TBL * qtbl = cinfo->quant_tbl_ptrs[index];
    int prec;
    int i;

    if (qtbl == NULL)
        ERREXIT1(cinfo, JERR_NO_QUANT_TABLE, index);

    prec = 0;
    for (i = 0; i < DCTSIZE2; i++) {
        if (qtbl->quantval[i] > 255)
            prec = 1;
    }
}

```



```

if (! qtbl->sent_table) {
    emit_marker(cinfo, M_DQT);

    emit_2bytes(cinfo, prec ? DCTSIZE2*2 + 1 + 2 : DCTSIZE2 + 1 + 2);

    emit_byte(cinfo, index + (prec<<4));

    for (i = 0; i < DCTSIZE2; i++) {
        /* The table entries must be emitted in zigzag order. */
        unsigned int qval = qtbl->quantval[jpeg_natural_order[i]];
        if (prec)
            emit_byte(cinfo, (int) (qval >> 8));
        emit_byte(cinfo, (int) (qval & 0xFF));
    }

    qtbl->sent_table = TRUE;
}

return prec;
}

LOCAL(void)
emit_dht (j_compress_ptr cinfo, int index, boolean is_ac)
/* Emit a DHT marker */
{
    JHUFF_TBL * htbl;
    int length, i;

    if (is_ac) {
        htbl = cinfo->ac_huff_tbl_ptrs[index];
        index += 0x10; /* output index has AC bit set */
    } else {
        htbl = cinfo->dc_huff_tbl_ptrs[index];
    }

    if (htbl == NULL)
        ERREXIT1(cinfo, JERR_NO_HUFF_TABLE, index);

    if (! htbl->sent_table) {
        emit_marker(cinfo, M_DHT);

        length = 0;
        for (i = 1; i <= 16; i++)
            length += htbl->bits[i];

        emit_2bytes(cinfo, length + 2 + 1 + 16);
        emit_byte(cinfo, index);

        for (i = 1; i <= 16; i++)
            emit_byte(cinfo, htbl->bits[i]);

        for (i = 0; i < length; i++)
            emit_byte(cinfo, htbl->huffval[i]);

        htbl->sent_table = TRUE;
    }
}

LOCAL(void)
emit_dac (j_compress_ptr cinfo)
/* Emit a DAC marker */
/* Since the useful info is so small, we want to emit all the tables in */
/* one DAC marker. Therefore this routine does its own scan of the table. */
{
#ifdef C_ARITH_CODING_SUPPORTED
    char dc_in_use[NUM_ARITH_TBLS];
    char ac_in_use[NUM_ARITH_TBLS];
    int length, i;
    jpeg_component_info *comp_ptr;

    for (i = 0; i < NUM_ARITH_TBLS; i++)
        dc_in_use[i] = ac_in_use[i] = 0;

    for (i = 0; i < cinfo->comps_in_scan; i++) {
        comp_ptr = cinfo->cur_comp_info[i];
        dc_in_use[comp_ptr->dc_tbl_no] = 1;
        ac_in_use[comp_ptr->ac_tbl_no] = 1;
    }
#endif
}

```

```

length = 0;
for (i = 0; i < NUM_ARITH_TBLS; i++)
    length += dc_in_use[i] + ac_in_use[i];

emit_marker(cinfo, M_DAC);

emit_2bytes(cinfo, length*2 + 2);

for (i = 0; i < NUM_ARITH_TBLS; i++) {
    if (dc_in_use[i]) {
        emit_byte(cinfo, i);
        emit_byte(cinfo, cinfo->arith_dc_L[i] + (cinfo->arith_dc_U[i]<<4));
    }
    if (ac_in_use[i]) {
        emit_byte(cinfo, i + 0x10);
        emit_byte(cinfo, cinfo->arith_ac_K[i]);
    }
}
#endif /* C_ARITH_CODING_SUPPORTED */
}

LOCAL(void)
emit_dri (j_compress_ptr cinfo)
/* Emit a DRI marker */
{
    emit_marker(cinfo, M_DRI);

    emit_2bytes(cinfo, 4); /* fixed length */

    emit_2bytes(cinfo, (int) cinfo->restart_interval);
}

LOCAL(void)
emit_sof (j_compress_ptr cinfo, JPEG_MARKER code)
/* Emit a SOF marker */
{
    int ci;
    jpeg_component_info *comp_ptr;

    emit_marker(cinfo, code);

    emit_2bytes(cinfo, 3 * cinfo->num_components + 2 + 5 + 1); /* length */

    /* Make sure image isn't bigger than SOF field can handle */
    if ((long) cinfo->image_height > 65535L ||
        (long) cinfo->image_width > 65535L)
        ERREXIT1(cinfo, JERR_IMAGE_TOO_BIG, (unsigned int) 65535);

    emit_byte(cinfo, cinfo->data_precision);
    emit_2bytes(cinfo, (int) cinfo->image_height);
    emit_2bytes(cinfo, (int) cinfo->image_width);

    emit_byte(cinfo, cinfo->num_components);

    for (ci = 0, comp_ptr = cinfo->comp_info; ci < cinfo->num_components;
         ci++, comp_ptr++) {
        emit_byte(cinfo, comp_ptr->component_id);
        emit_byte(cinfo, (comp_ptr->h_samp_factor << 4) + comp_ptr->v_samp_factor);
        emit_byte(cinfo, comp_ptr->quant_tbl_no);
    }
}

LOCAL(void)
emit_sos (j_compress_ptr cinfo)
/* Emit a SOS marker */
{
    int i, td, ta;
    jpeg_component_info *comp_ptr;

    emit_marker(cinfo, M_SOS);

    emit_2bytes(cinfo, 2 * cinfo->comps_in_scan + 2 + 1 + 3); /* length */

    emit_byte(cinfo, cinfo->comps_in_scan);

    for (i = 0; i < cinfo->comps_in_scan; i++) {

```

```

comp_ptr = cinfo->cur_comp_ptr[comp_ptr->component_id];
emit_byte(cinfo, comp_ptr->dc_tbl_no);
td = comp_ptr->dc_tbl_no;
ta = comp_ptr->ac_tbl_no;
if (cinfo->progressive_mode) {
    /* Progressive mode: only DC or only AC tables are used in one scan;
     * furthermore, Huffman coding of DC refinement uses no table at all.
     * We emit 0 for unused field(s); this is recommended by the P&M text
     * but does not seem to be specified in the standard.
     */
    if (cinfo->Ss == 0) {
        ta = 0;          /* DC scan */
        if (cinfo->Ah != 0 && !cinfo->arith_code)
            td = 0;      /* no DC table either */
        else {
            td = 0;      /* AC scan */
        }
    }
    emit_byte(cinfo, (td << 4) + ta);
}

emit_byte(cinfo, cinfo->Ss);
emit_byte(cinfo, cinfo->Se);
emit_byte(cinfo, (cinfo->Ah << 4) + cinfo->Al);
}

```

```

LOCAL(void)
emit_jfif_app0 (j_compress_ptr cinfo)
/* Emit a JFIF-compliant APP0 marker */
{
    /*
     * Length of APP0 block (2 bytes)
     * Block ID (4 bytes - ASCII "JFIF")
     * Zero byte (1 byte to terminate the ID string)
     * Version Major, Minor (2 bytes - major first)
     * Units (1 byte - 0x00 = none, 0x01 = inch, 0x02 = cm)
     * Xdpi (2 bytes - dots per unit horizontal)
     * Ydpi (2 bytes - dots per unit vertical)
     * Thumbnail X size (1 byte)
     * Thumbnail Y size (1 byte)
     */
    emit_marker(cinfo, M_APP0);
    emit_2bytes(cinfo, 2 + 4 + 1 + 2 + 1 + 2 + 2 + 1 + 1); /* length */
    emit_byte(cinfo, 0x4A); /* Identifier: ASCII "JFIF" */
    emit_byte(cinfo, 0x46);
    emit_byte(cinfo, 0x49);
    emit_byte(cinfo, 0x46);
    emit_byte(cinfo, 0);
    emit_byte(cinfo, cinfo->JFIF_major_version); /* Version fields */
    emit_byte(cinfo, cinfo->JFIF_minor_version);
    emit_byte(cinfo, cinfo->density_unit); /* Pixel size information */
    emit_2bytes(cinfo, (int) cinfo->X_density);
    emit_2bytes(cinfo, (int) cinfo->Y_density);
    emit_byte(cinfo, 0); /* No thumbnail image */
    emit_byte(cinfo, 0);
}

```

```

LOCAL(void)
emit_adobe_app14 (j_compress_ptr cinfo)
/* Emit an Adobe APP14 marker */
{
    /*
     * Length of APP14 block (2 bytes)
     * Block ID (5 bytes - ASCII "Adobe")
     * Version Number (2 bytes - currently 100)
     * Flags0 (2 bytes - currently 0)
     * Flags1 (2 bytes - currently 0)
     * Color transform (1 byte)
     *
     * Although Adobe TN 5116 mentions Version = 101, all the Adobe files
     * now in circulation seem to use Version = 100, so that's what we write.
     *
     * We write the color transform byte as 1 if the JPEG color space is
     * YCbCr, 2 if it's YCCk, 0 otherwise. Adobe's definition has to do with
     * whether the encoder performed a transformation, which is pretty useless.
     */
}

```

```

*/
emit_marker(cinfo, M_APP14),

emit_2bytes(cinfo, 2 + 5 + 2 + 2 + 2 + 1); /* length */

emit_byte(cinfo, 0x41); /* Identifier: ASCII "Adobe" */
emit_byte(cinfo, 0x64);
emit_byte(cinfo, 0x6F);
emit_byte(cinfo, 0x62);
emit_byte(cinfo, 0x65);
emit_2bytes(cinfo, 100); /* Version */
emit_2bytes(cinfo, 0); /* Flags0 */
emit_2bytes(cinfo, 0); /* Flags1 */
switch (cinfo->jpeg_color_space) {
case JCS_YCbCr:
    emit_byte(cinfo, 1); /* Color transform = 1 */
    break;
case JCS_YCCK:
    emit_byte(cinfo, 2); /* Color transform = 2 */
    break;
default:
    emit_byte(cinfo, 0); /* Color transform = 0 */
    break;
}
}

/*
 * These routines allow writing an arbitrary marker with parameters.
 * The only intended use is to emit COM or APPn markers after calling
 * write_file_header and before calling write_frame_header.
 * Other uses are not guaranteed to produce desirable results.
 * Counting the parameter bytes properly is the caller's responsibility.
 */

METHODDEF(void)
write_marker_header (j_compress_ptr cinfo, int marker, unsigned int datalen)
/* Emit an arbitrary marker header */
{
    if (datalen > (unsigned int) 65533) /* safety check */
        ERREXIT(cinfo, JERR_BAD_LENGTH);
    emit_marker(cinfo, (JPEG_MARKER) marker);
    emit_2bytes(cinfo, (int) (datalen + 2)); /* total length */
}

METHODDEF(void)
write_marker_byte (j_compress_ptr cinfo, int val)
/* Emit one byte of marker parameters following write_marker_header */
{
    emit_byte(cinfo, val);
}

/*
 * Write datastream header.
 * This consists of an SOI and optional APPn markers.
 * We recommend use of the JFIF marker, but not the Adobe marker,
 * when using YCbCr or grayscale data. The JFIF marker should NOT
 * be used for any other JPEG colorspace. The Adobe marker is helpful
 * to distinguish RGB, CMYK, and YCCK colorspaces.
 * Note that an application can write additional header markers after
 * jpeg_start_compress returns.
 */

METHODDEF(void)
write_file_header (j_compress_ptr cinfo)
{
    my_marker_ptr marker = (my_marker_ptr) cinfo->marker;

    emit_marker(cinfo, M_SOI); /* first the SOI */

    /* SOI is defined to reset restart interval to 0 */
    marker->last_restart_interval = 0;

    if (cinfo->write_JFIF_header) /* next an optional JFIF APP0 */
        emit_jfif_app0(cinfo);
    if (cinfo->write_Adobe_marker) /* next an optional Adobe APP14 */

```

```

    emit_adobe_appl4(cinfo);
}

/*
 * Write frame header.
 * This consists of DQT and SOFn markers.
 * Note that we do not emit the SOF until we have emitted the DQT(s).
 * This avoids compatibility problems with incorrect implementations that
 * try to error-check the quant table numbers as soon as they see the SOF.
 */

METHODDEF(void)
write_frame_header (j_compress_ptr cinfo)
{
    int ci, prec;
    boolean is_baseline;
    jpeg_component_info *comp_ptr;

    /* Emit DQT for each quantization table.
     * Note that emit_dqt() suppresses any duplicate tables.
     */
    prec = 0;
    for (ci = 0, comp_ptr = cinfo->comp_info; ci < cinfo->num_components;
         ci++, comp_ptr++) {
        prec += emit_dqt(cinfo, comp_ptr->quant_tbl_no);
    }
    /* now prec is nonzero iff there are any 16-bit quant tables. */

    /* Check for a non-baseline specification.
     * Note we assume that Huffman table numbers won't be changed later.
     */
    if (cinfo->arith_code || cinfo->progressive_mode ||
        cinfo->data_precision != 8) {
        is_baseline = FALSE;
    } else {
        is_baseline = TRUE;
        for (ci = 0, comp_ptr = cinfo->comp_info; ci < cinfo->num_components;
             ci++, comp_ptr++) {
            if (comp_ptr->dc_tbl_no > 1 || comp_ptr->ac_tbl_no > 1)
                is_baseline = FALSE;
        }
        if (prec && is_baseline) {
            is_baseline = FALSE;
            /* If it's baseline except for quantizer size, warn the user */
            TRACEMS(cinfo, 0, JTRC_16BIT_TABLES);
        }
    }

    /* Emit the proper SOF marker */
    if (cinfo->arith_code) {
        emit_sof(cinfo, M_SOF9); /* SOF code for arithmetic coding */
    } else {
        if (cinfo->progressive_mode)
            emit_sof(cinfo, M_SOF2); /* SOF code for progressive Huffman */
        else if (is_baseline)
            emit_sof(cinfo, M_SOF0); /* SOF code for baseline implementation */
        else
            emit_sof(cinfo, M_SOF1); /* SOF code for non-baseline Huffman file */
    }
}

/*
 * Write scan header.
 * This consists of DHT or DAC markers, optional DRI, and SOS.
 * Compressed data will be written following the SOS.
 */

METHODDEF(void)
write_scan_header (j_compress_ptr cinfo)
{
    my_marker_ptr marker = (my_marker_ptr) cinfo->marker;
    int i;
    jpeg_component_info *comp_ptr;

    if (cinfo->arith_code) {
        /* Emit arith conditioning info. We may have some duplication
         * if the file has multiple scans, but it's so small it's hardly
         * worth worrying about.

```

```

    */
    emit_dac(cinfo);
} else {
    /* Emit Huffman tables.
     * Note that emit_dht() suppresses any duplicate tables.
     */
    for (i = 0; i < cinfo->comps_in_scan; i++) {
        compptr = cinfo->cur_comp_info[i];
        if (cinfo->progressive_mode) {
            /* Progressive mode: only DC or only AC tables are used in one scan */
            if (cinfo->Ss == 0) {
                if (cinfo->Ah == 0) /* DC needs no table for refinement scan */
                    emit_dht(cinfo, compptr->dc_tbl_no, FALSE);
            } else {
                emit_dht(cinfo, compptr->ac_tbl_no, TRUE);
            }
        } else {
            /* Sequential mode: need both DC and AC tables */
            emit_dht(cinfo, compptr->dc_tbl_no, FALSE);
            emit_dht(cinfo, compptr->ac_tbl_no, TRUE);
        }
    }
}

/* Emit DRI if required --- note that DRI value could change for each scan.
 * We avoid wasting space with unnecessary DRIs, however.
 */
if (cinfo->restart_interval != marker->last_restart_interval) {
    emit_dri(cinfo);
    marker->last_restart_interval = cinfo->restart_interval;
}

emit_sos(cinfo);

/* Write datastream trailer.
 */
METHODDEF(void)
write_file_trailer (j_compress_ptr cinfo)
{
    emit_marker(cinfo, M_EOI);

    /* Write an abbreviated table-specification datastream.
     * This consists of SOI, DQT and DHT tables, and EOI.
     * Any table that is defined and not marked sent_table = TRUE will be
     * emitted. Note that all tables will be marked sent_table = TRUE at exit.
     */
    METHODDEF(void)
    write_tables_only (j_compress_ptr cinfo)
    {
        int i;

        emit_marker(cinfo, M_SOI);

        for (i = 0; i < NUM_QUANT_TBLS; i++) {
            if (cinfo->quant_tbl_ptrs[i] != NULL)
                (void) emit_dqt(cinfo, i);
        }

        if (! cinfo->arith_code) {
            for (i = 0; i < NUM_HUFF_TBLS; i++) {
                if (cinfo->dc_huff_tbl_ptrs[i] != NULL)
                    emit_dht(cinfo, i, FALSE);
                if (cinfo->ac_huff_tbl_ptrs[i] != NULL)
                    emit_dht(cinfo, i, TRUE);
            }
        }

        emit_marker(cinfo, M_EOI);
    }
}

/*

```



```

/*
 * jcmaster.c
 *
 * Copyright (C) 1991-1997, Thomas G. Lane.
 * This file is part of the Independent JPEG Group's software.
 * For conditions of distribution and use, see the accompanying README file.
 *
 * This file contains master control logic for the JPEG compressor.
 * These routines are concerned with parameter validation, initial setup,
 * and inter-pass control (determining the number of passes and the work
 * to be done in each pass).
 */

```

```

#define JPEG_INTERNALS
#include "jinclude.h"
#include "jpeglib.h"

```

```

/* Private state */

```

```

typedef enum {
    main_pass,          /* input data, also do first output step */
    huff_opt_pass,      /* Huffman code optimization pass */
    output_pass         /* data output pass */
} c_pass_type;

```

```

typedef struct {
    struct jpeg_comp_master pub; /* public fields */

    c_pass_type pass_type;      /* the type of the current pass */

    int pass_number;           /* # of passes completed */
    int total_passes;         /* total # of passes needed */

    int scan_number;          /* current index in scan_info[] */
} my_comp_master;

```

```

typedef my_comp_master * my_master_ptr;

```

```

/*
 * Support routines that do various essential calculations.
 */

```

```

LOCAL(void)
initial_setup (j_compress_ptr cinfo)
/* Do computations that are needed before master selection phase */
{
    int ci;
    jpeg_component_info *comp_ptr;
    long samplesperrow;
    JDIMENSION jd_samplesperrow;

    /* Sanity check on image dimensions */
    if (cinfo->image_height <= 0 || cinfo->image_width <= 0
        || cinfo->num_components <= 0 || cinfo->input_components <= 0)
        ERREXIT(cinfo, JERR_EMPTY_IMAGE);

    /* Make sure image isn't bigger than I can handle */
    if ((long) cinfo->image_height > (long) JPEG_MAX_DIMENSION ||
        (long) cinfo->image_width > (long) JPEG_MAX_DIMENSION)
        ERREXIT1(cinfo, JERR_IMAGE_TOO_BIG, (unsigned int) JPEG_MAX_DIMENSION);

    /* Width of an input scanline must be representable as JDIMENSION. */
    samplesperrow = (long) cinfo->image_width * (long) cinfo->input_components;
    jd_samplesperrow = (JDIMENSION) samplesperrow;
    if ((long) jd_samplesperrow != samplesperrow)
        ERREXIT(cinfo, JERR_WIDTH_OVERFLOW);

    /* For now, precision must match compiled-in value... */
    if (cinfo->data_precision != BITS_IN_JSAMPLE)
        ERREXIT1(cinfo, JERR_BAD_PRECISION, cinfo->data_precision);

    /* Check that number of components won't exceed internal array sizes */
    if (cinfo->num_components > MAX_COMPONENTS)
        ERREXIT2(cinfo, JERR_COMPONENT_COUNT, cinfo->num_components,
            MAX_COMPONENTS);

    /* Compute maximum sampling factors; check factor validity */
    cinfo->max_h_samp_factor = 1;

```



```

cinfo->max_v_samp_factor = 1;
for (ci = 0, compptr = cinfo->comp_info; ci < cinfo->num_components;
    ci++, compptr++) {
    if (compptr->h_samp_factor <= 0 || compptr->h_samp_factor > MAX_SAMP_FACTOR ||
        compptr->v_samp_factor <= 0 || compptr->v_samp_factor > MAX_SAMP_FACTOR)
        ERREXIT(cinfo, JERR_BAD_SAMPLING);
    cinfo->max_h_samp_factor = MAX(cinfo->max_h_samp_factor,
        compptr->h_samp_factor);
    cinfo->max_v_samp_factor = MAX(cinfo->max_v_samp_factor,
        compptr->v_samp_factor);
}

/* Compute dimensions of components */
for (ci = 0, compptr = cinfo->comp_info; ci < cinfo->num_components;
    ci++, compptr++) {
    /* Fill in the correct component_index value; don't rely on application */
    compptr->component_index = ci;
    /* For compression, we never do DCT scaling. */
    compptr->DCT_scaled_size = DCTSIZE;
    /* Size in DCT blocks */
    compptr->width_in_blocks = (JDIMENSION)
        jdiv_round_up((long) cinfo->image_width * (long) compptr->h_samp_factor,
            (long) (cinfo->max_h_samp_factor * DCTSIZE));
    compptr->height_in_blocks = (JDIMENSION)
        jdiv_round_up((long) cinfo->image_height * (long) compptr->v_samp_factor,
            (long) (cinfo->max_v_samp_factor * DCTSIZE));
    /* Size in samples */
    compptr->downsampled_width = (JDIMENSION)
        jdiv_round_up((long) cinfo->image_width * (long) compptr->h_samp_factor,
            (long) cinfo->max_h_samp_factor);
    compptr->downsampled_height = (JDIMENSION)
        jdiv_round_up((long) cinfo->image_height * (long) compptr->v_samp_factor,
            (long) cinfo->max_v_samp_factor);
    /* Mark component needed (this flag isn't actually used for compression) */
    compptr->component_needed = TRUE;

    /* Compute number of fully interleaved MCU rows (number of times that
     * main controller will call coefficient controller).
     */
    cinfo->total_iMCU_rows = (JDIMENSION)
        jdiv_round_up((long) cinfo->image_height,
            (long) (cinfo->max_v_samp_factor * DCTSIZE));
}

#ifdef C_MULTISCAN_FILES_SUPPORTED
LOCAL(void)
validate_script (j_compress_ptr cinfo)
/* Verify that the scan script in cinfo->scan_info[] is valid; also
 * determine whether it uses progressive JPEG, and set cinfo->progressive_mode.
 */
{
    const jpeg_scan_info * scanptr;
    int scanno, ncomps, ci, coefi, thisi;
    int Ss, Se, Ah, Al;
    boolean component_sent[MAX_COMPONENTS];
#ifdef C_PROGRESSIVE_SUPPORTED
    int * last_bitpos_ptr;
    int last_bitpos[MAX_COMPONENTS][DCTSIZE2];
    /* -1 until that coefficient has been seen; then last Al for it */
#endif
    #endif

    if (cinfo->num_scans <= 0)
        ERREXIT1(cinfo, JERR_BAD_SCAN_SCRIPT, 0);

    /* For sequential JPEG, all scans must have Ss=0, Se=DCTSIZE2-1;
     * for progressive JPEG, no scan can have this.
     */
    scanptr = cinfo->scan_info;
    if (scanptr->Ss != 0 || scanptr->Se != DCTSIZE2-1) {
#ifdef C_PROGRESSIVE_SUPPORTED
        cinfo->progressive_mode = TRUE;
        last_bitpos_ptr = & last_bitpos[0][0];
        for (ci = 0; ci < cinfo->num_components; ci++)
            for (coefi = 0; coefi < DCTSIZE2; coefi++)
                *last_bitpos_ptr++ = -1;
#else
        ERREXIT(cinfo, JERR_NOT_COMPILED);
#endif
    }
}

```

```

#endif
    } else {
        cinfo->progressive_mode = FALSE;
        for (ci = 0; ci < cinfo->num_components; ci++)
            component_sent[ci] = FALSE;
    }

    for (scanno = 1; scanno <= cinfo->num_scans; scanptr++, scanno++) {
        /* Validate component indexes */
        ncomps = scanptr->comps_in_scan;
        if (ncomps <= 0 || ncomps > MAX_COMPS_IN_SCAN)
            ERREXIT2(cinfo, JERR_COMPONENT_COUNT, ncomps, MAX_COMPS_IN_SCAN);
        for (ci = 0; ci < ncomps; ci++) {
            thisi = scanptr->component_index[ci];
            if (thisi < 0 || thisi >= cinfo->num_components)
                ERREXIT1(cinfo, JERR_BAD_SCAN_SCRIPT, scanno);
            /* Components must appear in SOF order within each scan */
            if (ci > 0 && thisi <= scanptr->component_index[ci-1])
                ERREXIT1(cinfo, JERR_BAD_SCAN_SCRIPT, scanno);
        }
        /* Validate progression parameters */
        Ss = scanptr->Ss;
        Se = scanptr->Se;
        Ah = scanptr->Ah;
        Al = scanptr->Al;
        if (cinfo->progressive_mode) {
#ifdef C_PROGRESSIVE_SUPPORTED
            /* The JPEG spec simply gives the ranges 0..13 for Ah and Al, but that
             * seems wrong: the upper bound ought to depend on data precision.
             * Perhaps they really meant 0..N+1 for N-bit precision.
             * Here we allow 0..10 for 8-bit data; Al larger than 10 results in
             * out-of-range reconstructed DC values during the first DC scan,
             * which might cause problems for some decoders.
             */
            #if BITS_IN_JSAMPLE == 8
            #define MAX_AH_AL 10
            #else
            #define MAX_AH_AL 13
            #endif
            if (Ss < 0 || Ss >= DCTSIZE2 || Se < Ss || Se >= DCTSIZE2 ||
                Ah < 0 || Ah > MAX_AH_AL || Al < 0 || Al > MAX_AH_AL)
                ERREXIT1(cinfo, JERR_BAD_PROG_SCRIPT, scanno);
            if (Ss == 0) {
                if (Se != 0) /* DC and AC together not OK */
                    ERREXIT1(cinfo, JERR_BAD_PROG_SCRIPT, scanno);
            } else {
                if (ncomps != 1) /* AC scans must be for only one component */
                    ERREXIT1(cinfo, JERR_BAD_PROG_SCRIPT, scanno);
            }
            for (ci = 0; ci < ncomps; ci++) {
                last_bitpos_ptr = & last_bitpos[scanptr->component_index[ci]][0];
                if (Ss != 0 && last_bitpos_ptr[0] < 0) /* AC without prior DC scan */
                    ERREXIT1(cinfo, JERR_BAD_PROG_SCRIPT, scanno);
                for (coefi = Ss; coefi <= Se; coefi++) {
                    if (last_bitpos_ptr[coefi] < 0) {
                        /* first scan of this coefficient */
                        if (Ah != 0)
                            ERREXIT1(cinfo, JERR_BAD_PROG_SCRIPT, scanno);
                    } else {
                        /* not first scan */
                        if (Ah != last_bitpos_ptr[coefi] || Al != Ah-1)
                            ERREXIT1(cinfo, JERR_BAD_PROG_SCRIPT, scanno);
                    }
                    last_bitpos_ptr[coefi] = Al;
                }
            }
        }
    }
#endif
    } else {
        /* For sequential JPEG, all progression parameters must be these: */
        if (Ss != 0 || Se != DCTSIZE2-1 || Ah != 0 || Al != 0)
            ERREXIT1(cinfo, JERR_BAD_PROG_SCRIPT, scanno);
        /* Make sure components are not sent twice */
        for (ci = 0; ci < ncomps; ci++) {
            thisi = scanptr->component_index[ci];
            if (component_sent[thisi])
                ERREXIT1(cinfo, JERR_BAD_SCAN_SCRIPT, scanno);
            component_sent[thisi] = TRUE;
        }
    }
}

```

```

/* Now verify that everything got sent. */
if (cinfo->progressive_mode) {
#ifdef C_PROGRESSIVE_SUPPORTED
/* For progressive mode, we only check that at least some DC data
 * got sent for each component; the spec does not require that all bits
 * of all coefficients be transmitted. Would it be wiser to enforce
 * transmission of all coefficient bits??
 */
for (ci = 0; ci < cinfo->num_components; ci++) {
    if (last_bitpos[ci][0] < 0)
        ERREXIT(cinfo, JERR_MISSING_DATA);
}
#endif
} else {
for (ci = 0; ci < cinfo->num_components; ci++) {
    if (! component_sent[ci])
        ERREXIT(cinfo, JERR_MISSING_DATA);
}
}
}

#endif /* C_MULTISCAN_FILES_SUPPORTED */

```

```

LOCAL(void)
select_scan_parameters (j_compress_ptr cinfo)
/* Set up the scan parameters for the current scan */
{
    int ci;

#ifdef C_MULTISCAN_FILES_SUPPORTED
    if (cinfo->scan_info != NULL) {
        /* Prepare for current scan --- the script is already validated */
        my_master_ptr master = (my_master_ptr) cinfo->master;
        const jpeg_scan_info * scanptr = cinfo->scan_info + master->scan_number;

        cinfo->comps_in_scan = scanptr->comps_in_scan;
        for (ci = 0; ci < scanptr->comps_in_scan; ci++) {
            cinfo->cur_comp_info[ci] =
                &cinfo->comp_info[scanptr->component_index[ci]];
        }
        cinfo->Ss = scanptr->Ss;
        cinfo->Se = scanptr->Se;
        cinfo->Ah = scanptr->Ah;
        cinfo->Al = scanptr->Al;
    }
    else
#endif
{
    /* Prepare for single sequential-JPEG scan containing all components */
    if (cinfo->num_components > MAX_COMPS_IN_SCAN)
        ERREXIT2(cinfo, JERR_COMPONENT_COUNT, cinfo->num_components,
            MAX_COMPS_IN_SCAN);
    cinfo->comps_in_scan = cinfo->num_components;
    for (ci = 0; ci < cinfo->num_components; ci++) {
        cinfo->cur_comp_info[ci] = &cinfo->comp_info[ci];
    }
    cinfo->Ss = 0;
    cinfo->Se = DCTSIZE2-1;
    cinfo->Ah = 0;
    cinfo->Al = 0;
}
}

```

```

LOCAL(void)
per_scan_setup (j_compress_ptr cinfo)
/* Do computations that are needed before processing a JPEG scan */
/* cinfo->comps_in_scan and cinfo->cur_comp_info[] are already set */
{
    int ci, mcublks, tmp;
    jpeg_component_info *comp_ptr;

    if (cinfo->comps_in_scan == 1) {
        /* Noninterleaved (single-component) scan */
        comp_ptr = cinfo->cur_comp_info[0];

        /* Overall image size in MCUs */

```

```

cinfo->MCUs_per_row = compptr->width_in_blocks;
cinfo->MCU_rows_in_scan = compptr->height_in_blocks;

/* For noninterleaved scan, always one block per MCU */
compptr->MCU_width = 1;
compptr->MCU_height = 1;
compptr->MCU_blocks = 1;
compptr->MCU_sample_width = DCTSIZE;
compptr->last_col_width = 1;
/* For noninterleaved scans, it is convenient to define last_row_height
 * as the number of block rows present in the last iMCU row.
 */
tmp = (int) (compptr->height_in_blocks % compptr->v_samp_factor);
if (tmp == 0) tmp = compptr->v_samp_factor;
compptr->last_row_height = tmp;

/* Prepare array describing MCU composition */
cinfo->blocks_in_MCU = 1;
cinfo->MCU_membership[0] = 0;

) else {

/* Interleaved (multi-component) scan */
if (cinfo->comps_in_scan <= 0 || cinfo->comps_in_scan > MAX_COMPS_IN_SCAN)
    ERREXIT2(cinfo, JERR_COMPONENT_COUNT, cinfo->comps_in_scan,
             MAX_COMPS_IN_SCAN);

/* Overall image size in MCUs */
cinfo->MCUs_per_row = (JDIMENSION)
    jdiv_round_up((long) cinfo->image_width,
                  (long) (cinfo->max_h_samp_factor*DCTSIZE));
cinfo->MCU_rows_in_scan = (JDIMENSION)
    jdiv_round_up((long) cinfo->image_height,
                  (long) (cinfo->max_v_samp_factor*DCTSIZE));

cinfo->blocks_in_MCU = 0;

for (ci = 0; ci < cinfo->comps_in_scan; ci++) {
    compptr = cinfo->cur_comp_info[ci];
    /* Sampling factors give # of blocks of component in each MCU */
    compptr->MCU_width = compptr->h_samp_factor;
    compptr->MCU_height = compptr->v_samp_factor;
    compptr->MCU_blocks = compptr->MCU_width * compptr->MCU_height;
    compptr->MCU_sample_width = compptr->MCU_width * DCTSIZE;
    /* Figure number of non-dummy blocks in last MCU column & row */
    tmp = (int) (compptr->width_in_blocks % compptr->MCU_width);
    if (tmp == 0) tmp = compptr->MCU_width;
    compptr->last_col_width = tmp;
    tmp = (int) (compptr->height_in_blocks % compptr->MCU_height);
    if (tmp == 0) tmp = compptr->MCU_height;
    compptr->last_row_height = tmp;
    /* Prepare array describing MCU composition */
    mcublk = compptr->MCU_blocks;
    if (cinfo->blocks_in_MCU + mcublk > C_MAX_BLOCKS_IN_MCU)
        ERREXIT(cinfo, JERR_BAD_MCU_SIZE);
    while (mcublk-- > 0) {
        cinfo->MCU_membership[cinfo->blocks_in_MCU++] = ci;
    }
}

/* Convert restart specified in rows to actual MCU count. */
/* Note that count must fit in 16 bits, so we provide limiting. */
if (cinfo->restart_in_rows > 0) {
    long nominal = (long) cinfo->restart_in_rows * (long) cinfo->MCUs_per_row;
    cinfo->restart_interval = (unsigned int) MIN(nominal, 65535L);
}

/*
 * Per-pass setup.
 * This is called at the beginning of each pass. We determine which modules
 * will be active during this pass and give them appropriate start_pass calls.
 * We also set is_last_pass to indicate whether any more passes will be
 * required.
 */

```

```

METHODDEF(void)

```

```

prepare_for_pass (j_compress_ptr cinfo)
{
    my_master_ptr master = (my_master_ptr) cinfo->master;

    switch (master->pass_type) {
    case main_pass:
        /* Initial pass: will collect input data, and do either Huffman
         * optimization or data output for the first scan.
         */
        select_scan_parameters(cinfo);
        per_scan_setup(cinfo);
        if (! cinfo->raw_data_in) {
            (*cinfo->cconvert->start_pass) (cinfo);
            (*cinfo->downsample->start_pass) (cinfo);
            (*cinfo->prep->start_pass) (cinfo, JBUF_PASS_THRU);
        }
        (*cinfo->fdct->start_pass) (cinfo);
        (*cinfo->entropy->start_pass) (cinfo, cinfo->optimize_coding);
        (*cinfo->coef->start_pass) (cinfo,
            (master->total_passes > 1 ?
             JBUF_SAVE_AND_PASS : JBUF_PASS_THRU));
        (*cinfo->main->start_pass) (cinfo, JBUF_PASS_THRU);
        if (cinfo->optimize_coding) {
            /* No immediate data output; postpone writing frame/scan headers */
            master->pub.call_pass_startup = FALSE;
        } else {
            /* Will write frame/scan headers at first jpeg_write_scanlines call */
            master->pub.call_pass_startup = TRUE;
        }
        break;
#ifdef ENTROPY_OPT_SUPPORTED
    case huff_opt_pass:
        /* Do Huffman optimization for a scan after the first one. */
        select_scan_parameters(cinfo);
        per_scan_setup(cinfo);
        if (cinfo->Ss != 0 || cinfo->Ah == 0 || cinfo->arith_code) {
            (*cinfo->entropy->start_pass) (cinfo, TRUE);
            (*cinfo->coef->start_pass) (cinfo, JBUF_CRANK_DEST);
            master->pub.call_pass_startup = FALSE;
            break;
        }
        /* Special case: Huffman DC refinement scans need no Huffman table
         * and therefore we can skip the optimization pass for them.
         */
        master->pass_type = output_pass;
        master->pass_number++;
        /* FALLTHROUGH */
#endif
    case output_pass:
        /* Do a data-output pass. */
        /* We need not repeat per-scan setup if prior optimization pass did it. */
        if (! cinfo->optimize_coding) {
            select_scan_parameters(cinfo);
            per_scan_setup(cinfo);
        }
        (*cinfo->entropy->start_pass) (cinfo, FALSE);
        (*cinfo->coef->start_pass) (cinfo, JBUF_CRANK_DEST);
        /* We emit frame/scan headers now */
        if (master->scan_number == 0)
            (*cinfo->marker->write_frame_header) (cinfo);
        (*cinfo->marker->write_scan_header) (cinfo);
        master->pub.call_pass_startup = FALSE;
        break;
    default:
        ERREXIT(cinfo, JERR_NOT_COMPILED);
    }

    master->pub.is_last_pass = (master->pass_number == master->total_passes-1);

    /* Set up progress monitor's pass info if present */
    if (cinfo->progress != NULL) {
        cinfo->progress->completed_passes = master->pass_number;
        cinfo->progress->total_passes = master->total_passes;
    }
}

/*
 * Special start-of-pass hook.
 * This is called by jpeg_write_scanlines if call_pass_startup is TRUE.

```

```

* In single-pass processing, need this hook because we don't want to
* write frame/scan headers during jpeg_start_compress; we want to let the
* application write COM markers, etc. between jpeg_start_compress and the
* jpeg_write_scanlines loop.
* In multi-pass processing, this routine is not used.
*/

```

```

METHODDEF(void)
pass_startup (j_compress_ptr cinfo)
{
    cinfo->master->call_pass_startup = FALSE; /* reset flag so call only once */

    (*cinfo->marker->write_frame_header) (cinfo);
    (*cinfo->marker->write_scan_header) (cinfo);
}

/*
 * Finish up at end of pass.
 */

```

```

METHODDEF(void)
finish_pass_master (j_compress_ptr cinfo)
{
    my_master_ptr master = (my_master_ptr) cinfo->master;

    /* The entropy coder always needs an end-of-pass call,
     * either to analyze statistics or to flush its output buffer.
     */
    (*cinfo->entropy->finish_pass) (cinfo);

    /* Update state for next pass */
    switch (master->pass_type) {
    case main_pass:
        /* next pass is either output of scan 0 (after optimization)
         * or output of scan 1 (if no optimization).
         */
        master->pass_type = output_pass;
        if (! cinfo->optimize_coding)
            master->scan_number++;
        break;
    case huff_opt_pass:
        /* next pass is always output of current scan */
        master->pass_type = output_pass;
        break;
    case output_pass:
        /* next pass is either optimization or output of next scan */
        if (cinfo->optimize_coding)
            master->pass_type = huff_opt_pass;
        master->scan_number++;
        break;
    }
    master->pass_number++;
}

/*
 * Initialize master compression control.
 */

```

```

GLOBAL(void)
jinit_c_master_control (j_compress_ptr cinfo, boolean transcode_only)
{
    my_master_ptr master;

    master = (my_master_ptr)
        (*cinfo->mem->alloc_small) ((j_common_ptr) cinfo, JPOOL_IMAGE,
            sizeof(my_comp_master));
    cinfo->master = (struct jpeg_comp_master *) master;
    master->pub.prepare_for_pass = prepare_for_pass;
    master->pub.pass_startup = pass_startup;
    master->pub.finish_pass = finish_pass_master;
    master->pub.is_last_pass = FALSE;

    /* Validate parameters, determine derived values */
    initial_setup(cinfo);

    if (cinfo->scan_info != NULL) {
#ifdef C_MULTISCAN_FILES_SUPPORTED

```



```

/*
 * jcomapi.c
 *
 * Copyright (C) 1994-1997, Thomas G. Lane.
 * This file is part of the Independent JPEG Group's software.
 * For conditions of distribution and use, see the accompanying README file.
 *
 * This file contains application interface routines that are used for both
 * compression and decompression.
 */

#define JPEG_INTERNALS
#include "jinclude.h"
#include "jpeglib.h"

/*
 * Abort processing of a JPEG compression or decompression operation,
 * but don't destroy the object itself.
 *
 * For this, we merely clean up all the nonpermanent memory pools.
 * Note that temp files (virtual arrays) are not allowed to belong to
 * the permanent pool, so we will be able to close all temp files here.
 * Closing a data source or destination, if necessary, is the application's
 * responsibility.
 */

GLOBAL(void)
jpeg_abort (j_common_ptr cinfo)
{
    int pool;

    /* Do nothing if called on a not-initialized or destroyed JPEG object. */
    if (cinfo->mem == NULL)
        return;

    /* Releasing pools in reverse order might help avoid fragmentation
     * with some (brain-damaged) malloc libraries.
     */
    for (pool = JPOOL_NUMPOOLS-1; pool > JPOOL_PERMANENT; pool--) {
        (*cinfo->mem->free_pool) (cinfo, pool);
    }

    /* Reset overall state for possible reuse of object */
    if (cinfo->is_decompressor) {
        cinfo->global_state = DSTATE_START;
        /* Try to keep application from accessing now-deleted marker list.
         * A bit kludgy to do it here, but this is the most central place.
         */
        ((j_decompress_ptr) cinfo)->marker_list = NULL;
    } else {
        cinfo->global_state = CSTATE_START;
    }
}

/*
 * Destruction of a JPEG object.
 *
 * Everything gets deallocated except the master jpeg_compress_struct itself
 * and the error manager struct. Both of these are supplied by the application
 * and must be freed, if necessary, by the application. (Often they are on
 * the stack and so don't need to be freed anyway.)
 * Closing a data source or destination, if necessary, is the application's
 * responsibility.
 */

GLOBAL(void)
jpeg_destroy (j_common_ptr cinfo)
{
    /* We need only tell the memory manager to release everything. */
    /* NB: mem pointer is NULL if memory mgr failed to initialize. */
    if (cinfo->mem != NULL)
        (*cinfo->mem->self_destruct) (cinfo);
    cinfo->mem = NULL; /* be safe if jpeg_destroy is called twice */
    cinfo->global_state = 0; /* mark it destroyed */
}

/*

```





```

/*
 * jcpparam.c
 *
 * Copyright (C) 1991-1998, Thomas G. Lane.
 * This file is part of the Independent JPEG Group's software.
 * For conditions of distribution and use, see the accompanying README file.
 *
 * This file contains optional default-setting code for the JPEG compressor.
 * Applications do not have to use this file, but those that don't use it
 * must know a lot more about the innards of the JPEG code.
 */

#define JPEG_INTERNALS
#include "jinclude.h"
#include "jpeglib.h"

/*
 * Quantization table setup routines
 */

GLOBAL(void)
jpeg_add_quant_table (j_compress_ptr cinfo, int which_tbl,
                     const unsigned int *basic_table,
                     int scale_factor, boolean force_baseline)
/* Define a quantization table equal to the basic_table times
 * a scale factor (given as a percentage).
 * If force_baseline is TRUE, the computed quantization table entries
 * are limited to 1..255 for JPEG baseline compatibility.
 */
{
  JQUANT_TBL ** qtblptr;
  int i;
  long temp;

  /* Safety check to ensure start_compress not called yet. */
  if (cinfo->global_state != CSTATE_START)
    ERREXIT1(cinfo, JERR_BAD_STATE, cinfo->global_state);

  if (which_tbl < 0 || which_tbl >= NUM_QUANT_TBLS)
    ERREXIT1(cinfo, JERR_DQT_INDEX, which_tbl);

  qtblptr = & cinfo->quant_tbl_ptrs[which_tbl];

  if (*qtblptr == NULL)
    *qtblptr = jpeg_alloc_quant_table((j_common_ptr) cinfo);

  for (i = 0; i < DCTSIZE2; i++) {
    temp = ((long) basic_table[i] * scale_factor + 50L) / 100L;
    /* limit the values to the valid range */
    if (temp <= 0L) temp = 1L;
    if (temp > 32767L) temp = 32767L; /* max quantizer needed for 12 bits */
    if (force_baseline && temp > 255L)
      temp = 255L; /* limit to baseline range if requested */
    (*qtblptr)->quantval[i] = (UINT16) temp;
  }

  /* Initialize sent_table FALSE so table will be written to JPEG file. */
  (*qtblptr)->sent_table = FALSE;
}

GLOBAL(void)
jpeg_set_linear_quality (j_compress_ptr cinfo, int scale_factor,
                        boolean force_baseline)
/* Set or change the 'quality' (quantization) setting, using default tables
 * and a straight percentage-scaling quality scale. In most cases it's better
 * to use jpeg_set_quality (below); this entry point is provided for
 * applications that insist on a linear percentage scaling.
 */
{
  /* These are the sample quantization tables given in JPEG spec section K.1.
   * The spec says that the values given produce "good" quality, and
   * when divided by 2, "very good" quality.
   */
  static const unsigned int std_luminance_quant_tbl[DCTSIZE2] = {
    16, 11, 10, 16, 24, 40, 51, 61,
    12, 12, 14, 19, 26, 58, 60, 55,
    14, 13, 16, 24, 40, 57, 69, 56,
    14, 17, 22, 29, 51, 87, 80, 62,
  }

```

```

18, 22, 37, 56, 68, 103, 77,
24, 35, 55, 64, 81, 113, 92,
49, 64, 78, 87, 103, 120, 101,
72, 92, 95, 98, 112, 100, 103, 99
);
static const unsigned int std_chrominance_quant_tbl[DCTSIZE2] = {
    17, 18, 24, 47, 99, 99, 99, 99,
    18, 21, 26, 66, 99, 99, 99, 99,
    24, 26, 56, 99, 99, 99, 99, 99,
    47, 66, 99, 99, 99, 99, 99, 99,
    99, 99, 99, 99, 99, 99, 99, 99,
    99, 99, 99, 99, 99, 99, 99, 99,
    99, 99, 99, 99, 99, 99, 99, 99,
    99, 99, 99, 99, 99, 99, 99, 99
};

/* Set up two quantization tables using the specified scaling */
jpeg_add_quant_table(cinfo, 0, std_luminance_quant_tbl,
    scale_factor, force_baseline);
jpeg_add_quant_table(cinfo, 1, std_chrominance_quant_tbl,
    scale_factor, force_baseline);
}

GLOBAL(int)
jpeg_quality_scaling (int quality)
/* Convert a user-specified quality rating to a percentage scaling factor
 * for an underlying quantization table, using our recommended scaling curve.
 * The input 'quality' factor should be 0 (terrible) to 100 (very good).
 */
{
    /* Safety limit on quality factor. Convert 0 to 1 to avoid zero divide. */
    if (quality <= 0) quality = 1;
    if (quality > 100) quality = 100;

    /* The basic table is used as-is (scaling 100) for a quality of 50.
     * Qualities 50..100 are converted to scaling percentage 200 - 2*Q;
     * note that at Q=100 the scaling is 0, which will cause jpeg_add_quant_table
     * to make all the table entries 1 (hence, minimum quantization loss).
     * Qualities 1..50 are converted to scaling percentage 5000/Q.
     */
    if (quality < 50)
        quality = 5000 / quality;
    else
        quality = 200 - quality*2;

    return quality;
}

GLOBAL(void)
jpeg_set_quality (j_compress_ptr cinfo, int quality, boolean force_baseline)
/* Set or change the 'quality' (quantization) setting, using default tables.
 * This is the standard quality-adjusting entry point for typical user
 * interfaces; only those who want detailed control over quantization tables
 * would use the preceding three routines directly.
 */
{
    /* Convert user 0-100 rating to percentage scaling */
    quality = jpeg_quality_scaling(quality);

    /* Set up standard quality tables */
    jpeg_set_linear_quality(cinfo, quality, force_baseline);
}

/*
 * Huffman table setup routines
 */

LOCAL(void)
add_huff_table (j_compress_ptr cinfo,
    JHUFF_TBL **htblptr, const UINT8 *bits, const UINT8 *val)
/* Define a Huffman table */
{
    int nsymbols, len;

    if (*htblptr == NULL)
        *htblptr = jpeg_alloc_huff_table((j_common_ptr) cinfo);
}

```

```

/* Copy the number-of-symbols-of-each-code-length counts */
MEMCOPY((*htblptr)->bits, 1, sizeof((*htblptr)->bits));

/* Validate the counts. We do this here mainly so we can copy the right
 * number of symbols from the val[] array, without risking marching off
 * the end of memory. jchuff.c will do a more thorough test later.
 */
nsymbols = 0;
for (len = 1; len <= 16; len++)
    nsymbols += bits[len];
if (nsymbols < 1 || nsymbols > 256)
    ERREXIT(cinfo, JERR_BAD_HUFF_TABLE);

MEMCOPY((*htblptr)->huffval, val, nsymbols * sizeof(UINT8));

/* Initialize sent_table FALSE so table will be written to JPEG file. */
(*htblptr)->sent_table = FALSE;
}

```

```

LOCAL(void)
std_huff_tables (j_compress_ptr cinfo)
/* Set up the standard Huffman tables (cf. JPEG standard section K.3) */
/* IMPORTANT: these are only valid for 8-bit data precision! */
{
    static const UINT8 bits_dc_luminance[17] =
        { /* 0-base */ 0, 0, 1, 5, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0 };
    static const UINT8 val_dc_luminance[] =
        { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 };

    static const UINT8 bits_dc_chrominance[17] =
        { /* 0-base */ 0, 0, 3, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0 };
    static const UINT8 val_dc_chrominance[] =
        { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 };

    static const UINT8 bits_ac_luminance[17] =
        { /* 0-base */ 0, 0, 2, 1, 3, 3, 2, 4, 3, 5, 5, 4, 4, 0, 0, 1, 0x7d };
    static const UINT8 val_ac_luminance[] =
        { 0x01, 0x02, 0x03, 0x00, 0x04, 0x11, 0x05, 0x12,
          0x21, 0x31, 0x41, 0x06, 0x13, 0x51, 0x61, 0x07,
          0x22, 0x71, 0x14, 0x32, 0x81, 0x91, 0xa1, 0x08,
          0x23, 0x42, 0xb1, 0xc1, 0x15, 0x52, 0xd1, 0xf0,
          0x24, 0x33, 0x62, 0x72, 0x82, 0x09, 0x0a, 0x16,
          0x17, 0x18, 0x19, 0x1a, 0x25, 0x26, 0x27, 0x28,
          0x29, 0x2a, 0x34, 0x35, 0x36, 0x37, 0x38, 0x39,
          0x3a, 0x43, 0x44, 0x45, 0x46, 0x47, 0x48, 0x49,
          0x4a, 0x53, 0x54, 0x55, 0x56, 0x57, 0x58, 0x59,
          0x5a, 0x63, 0x64, 0x65, 0x66, 0x67, 0x68, 0x69,
          0x6a, 0x73, 0x74, 0x75, 0x76, 0x77, 0x78, 0x79,
          0x7a, 0x83, 0x84, 0x85, 0x86, 0x87, 0x88, 0x89,
          0x8a, 0x92, 0x93, 0x94, 0x95, 0x96, 0x97, 0x98,
          0x99, 0x9a, 0xa2, 0xa3, 0xa4, 0xa5, 0xa6, 0xa7,
          0xa8, 0xa9, 0xaa, 0xb2, 0xb3, 0xb4, 0xb5, 0xb6,
          0xb7, 0xb8, 0xb9, 0xba, 0xc2, 0xc3, 0xc4, 0xc5,
          0xc6, 0xc7, 0xc8, 0xc9, 0xca, 0xd2, 0xd3, 0xd4,
          0xd5, 0xd6, 0xd7, 0xd8, 0xd9, 0xda, 0xe1, 0xe2,
          0xe3, 0xe4, 0xe5, 0xe6, 0xe7, 0xe8, 0xe9, 0xea,
          0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7, 0xf8,
          0xf9, 0xfa };

    static const UINT8 bits_ac_chrominance[17] =
        { /* 0-base */ 0, 0, 2, 1, 2, 4, 4, 3, 4, 7, 5, 4, 4, 0, 1, 2, 0x77 };
    static const UINT8 val_ac_chrominance[] =
        { 0x00, 0x01, 0x02, 0x03, 0x11, 0x04, 0x05, 0x21,
          0x31, 0x06, 0x12, 0x41, 0x51, 0x07, 0x61, 0x71,
          0x13, 0x22, 0x32, 0x81, 0x08, 0x14, 0x42, 0x91,
          0xa1, 0xb1, 0xc1, 0x09, 0x23, 0x33, 0x52, 0xf0,
          0x15, 0x62, 0x72, 0xd1, 0x0a, 0x16, 0x24, 0x34,
          0xe1, 0x25, 0xf1, 0x17, 0x18, 0x19, 0x1a, 0x26,
          0x27, 0x28, 0x29, 0x2a, 0x35, 0x36, 0x37, 0x38,
          0x39, 0x3a, 0x43, 0x44, 0x45, 0x46, 0x47, 0x48,
          0x49, 0x4a, 0x53, 0x54, 0x55, 0x56, 0x57, 0x58,
          0x59, 0x5a, 0x63, 0x64, 0x65, 0x66, 0x67, 0x68,
          0x69, 0x6a, 0x73, 0x74, 0x75, 0x76, 0x77, 0x78,
          0x79, 0x7a, 0x82, 0x83, 0x84, 0x85, 0x86, 0x87,
          0x88, 0x89, 0x8a, 0x92, 0x93, 0x94, 0x95, 0x96,
          0x97, 0x98, 0x99, 0x9a, 0xa2, 0xa3, 0xa4, 0xa5,
          0xa6, 0xa7, 0xa8, 0xa9, 0xaa, 0xb2, 0xb3, 0xb4,
          0xb5, 0xb6, 0xb7, 0xb8, 0xb9, 0xba, 0xc2, 0xc3,
          0xc4, 0xc5, 0xc6, 0xc7, 0xc8, 0xc9, 0xca, 0xd2,

```

```

0xd3, 0xd4, 0xd5, 0xd6, 0xd7, 0xd8, 0xd9, 0xda,
0xe2, 0xe3, 0xe4, 0xe5, 0xe6, 0xe7, 0xe8, 0xe9,
0xea, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7, 0xf8,
0xf9, 0xfa };

```

```

add_huff_table(cinfo, &cinfo->dc_huff_tbl_ptrs[0],
               bits_dc_luminance, val_dc_luminance);
add_huff_table(cinfo, &cinfo->ac_huff_tbl_ptrs[0],
               bits_ac_luminance, val_ac_luminance);
add_huff_table(cinfo, &cinfo->dc_huff_tbl_ptrs[1],
               bits_dc_chrominance, val_dc_chrominance);
add_huff_table(cinfo, &cinfo->ac_huff_tbl_ptrs[1],
               bits_ac_chrominance, val_ac_chrominance);
}

```

```

/*
 * Default parameter setup for compression.
 *
 * Applications that don't choose to use this routine must do their
 * own setup of all these parameters. Alternately, you can call this
 * to establish defaults and then alter parameters selectively. This
 * is the recommended approach since, if we add any new parameters,
 * your code will still work (they'll be set to reasonable defaults).
 */

```

```

GLOBAL(void)
jpeg_set_defaults (j_compress_ptr cinfo)
{
    int i;

    /* Safety check to ensure start_compress not called yet. */
    if (cinfo->global_state != CSTATE_START)
        ERREXIT1(cinfo, JERR_BAD_STATE, cinfo->global_state);

    /* Allocate comp_info array large enough for maximum component count.
     * Array is made permanent in case application wants to compress
     * multiple images at same param settings.
     */
    if (cinfo->comp_info == NULL)
        cinfo->comp_info = (jpeg_component_info *)
            (*cinfo->mem->alloc_small) ((j_common_ptr) cinfo, JPOOL_PERMANENT,
                                       MAX_COMPONENTS * SIZEOF(jpeg_component_info));

    /* Initialize everything not dependent on the color space */
    cinfo->data_precision = BITS_IN_JSAMPLE;
    /* Set up two quantization tables using default quality of 75 */
    jpeg_set_quality(cinfo, 75, TRUE);
    /* Set up two Huffman tables */
    std_huff_tables(cinfo);

    /* Initialize default arithmetic coding conditioning */
    for (i = 0; i < NUM_ARITH_TBLS; i++) {
        cinfo->arith_dc_L[i] = 0;
        cinfo->arith_dc_U[i] = 1;
        cinfo->arith_ac_K[i] = 5;
    }

    /* Default is no multiple-scan output */
    cinfo->scan_info = NULL;
    cinfo->num_scans = 0;

    /* Expect normal source image, not raw downsampled data */
    cinfo->raw_data_in = FALSE;

    /* Use Huffman coding, not arithmetic coding, by default */
    cinfo->arith_code = FALSE;

    /* By default, don't do extra passes to optimize entropy coding */
    cinfo->optimize_coding = FALSE;
    /* The standard Huffman tables are only valid for 8-bit data precision.
     * If the precision is higher, force optimization on so that usable
     * tables will be computed. This test can be removed if default tables
     * are supplied that are valid for the desired precision.
     */
    if (cinfo->data_precision > 8)
        cinfo->optimize_coding = TRUE;

    /* By default, use the simpler non-cosited sampling alignment */
}

```

```

cinfo->CCIR601_sampling = FALSE;

/* No input smoothing */
cinfo->smoothing_factor = 0;

/* DCT algorithm preference */
cinfo->dct_method = JDCT_DEFAULT;

/* No restart markers */
cinfo->restart_interval = 0;
cinfo->restart_in_rows = 0;

/* Fill in default JFIF marker parameters. Note that whether the marker
 * will actually be written is determined by jpeg_set_colorspace.
 *
 * By default, the library emits JFIF version code 1.01.
 * An application that wants to emit JFIF 1.02 extension markers should set
 * JFIF_minor_version to 2. We could probably get away with just defaulting
 * to 1.02, but there may still be some decoders in use that will complain
 * about that; saying 1.01 should minimize compatibility problems.
 */
cinfo->JFIF_major_version = 1; /* Default JFIF version = 1.01 */
cinfo->JFIF_minor_version = 1;
cinfo->density_unit = 0; /* Pixel size is unknown by default */
cinfo->X_density = 1; /* Pixel aspect ratio is square by default */
cinfo->Y_density = 1;

/* Choose JPEG colorspace based on input space, set defaults accordingly */
jpeg_default_colorspace(cinfo);
}

/* Select an appropriate JPEG colorspace for in_color_space.
 */
GLOBAL(void)
jpeg_default_colorspace (j_compress_ptr cinfo)
{
    switch (cinfo->in_color_space) {
        case JCS_GRAYSCALE:
            jpeg_set_colorspace(cinfo, JCS_GRAYSCALE);
            break;
        case JCS_RGB:
            jpeg_set_colorspace(cinfo, JCS_YCbCr);
            break;
        case JCS_YCbCr:
            jpeg_set_colorspace(cinfo, JCS_YCbCr);
            break;
        case JCS_CMYK:
            jpeg_set_colorspace(cinfo, JCS_CMYK); /* By default, no translation */
            break;
        case JCS_YCCK:
            jpeg_set_colorspace(cinfo, JCS_YCCK);
            break;
        case JCS_UNKNOWN:
            jpeg_set_colorspace(cinfo, JCS_UNKNOWN);
            break;
        default:
            ERREXIT(cinfo, JERR_BAD_IN_COLORSPACE);
    }
}

/*
 * Set the JPEG colorspace, and choose colorspace-dependent default values.
 */
GLOBAL(void)
jpeg_set_colorspace (j_compress_ptr cinfo, J_COLOR_SPACE colorspace)
{
    jpeg_component_info * compptr;
    int ci;

#define SET_COMP(index,id,hsamp,vsamp,quant,dctbl,actbl) \
    (compptr = &cinfo->comp_info[index], \
     compptr->component_id = (id), \
     compptr->h_samp_factor = (hsamp), \
     compptr->v_samp_factor = (vsamp), \

```

```

comp_ptr->quant_tbl_no = (quant), \
comp_ptr->dc_tbl_no = (dctbl), \
comp_ptr->ac_tbl_no = (actbl);

/* Safety check to ensure start_compress not called yet. */
if (cinfo->global_state != CSTATE_START)
    ERREXIT1(cinfo, JERR_BAD_STATE, cinfo->global_state);

/* For all colorspace, we use Q and Huff tables 0 for luminance components,
 * tables 1 for chrominance components.
 */

cinfo->jpeg_color_space = colorspace;

cinfo->write_JFIF_header = FALSE; /* No marker for non-JFIF colorspace */
cinfo->write_Adobe_marker = FALSE; /* write no Adobe marker by default */

switch (cinfo->jpeg_color_space) {
case JCS_GRAYSCALE:
    cinfo->write_JFIF_header = TRUE; /* Write a JFIF marker */
    cinfo->num_components = 1;
    /* JFIF specifies component ID 1 */
    SET_COMP(0, 1, 1, 1, 0, 0, 0);
    break;
case JCS_RGB:
    cinfo->write_Adobe_marker = TRUE; /* write Adobe marker to flag RGB */
    cinfo->num_components = 3;
    SET_COMP(0, 0x52 /* 'R' */, 1, 1, 0, 0, 0);
    SET_COMP(1, 0x47 /* 'G' */, 1, 1, 0, 0, 0);
    SET_COMP(2, 0x42 /* 'B' */, 1, 1, 0, 0, 0);
    break;
case JCS_YCbCr:
    cinfo->write_JFIF_header = TRUE; /* Write a JFIF marker */
    cinfo->num_components = 3;
    /* JFIF specifies component IDs 1,2,3 */
    /* We default to 2x2 subsamples of chrominance */
    SET_COMP(0, 1, 2, 2, 0, 0, 0);
    SET_COMP(1, 2, 1, 1, 1, 1, 1);
    SET_COMP(2, 3, 1, 1, 1, 1, 1);
    break;
case JCS_CMYK:
    cinfo->write_Adobe_marker = TRUE; /* write Adobe marker to flag CMYK */
    cinfo->num_components = 4;
    SET_COMP(0, 0x43 /* 'C' */, 1, 1, 0, 0, 0);
    SET_COMP(1, 0x4D /* 'M' */, 1, 1, 0, 0, 0);
    SET_COMP(2, 0x59 /* 'Y' */, 1, 1, 0, 0, 0);
    SET_COMP(3, 0x4B /* 'K' */, 1, 1, 0, 0, 0);
    break;
case JCS_YCCK:
    cinfo->write_Adobe_marker = TRUE; /* write Adobe marker to flag YCCK */
    cinfo->num_components = 4;
    SET_COMP(0, 1, 2, 2, 0, 0, 0);
    SET_COMP(1, 2, 1, 1, 1, 1, 1);
    SET_COMP(2, 3, 1, 1, 1, 1, 1);
    SET_COMP(3, 4, 2, 2, 0, 0, 0);
    break;
case JCS_UNKNOWN:
    cinfo->num_components = cinfo->input_components;
    if (cinfo->num_components < 1 || cinfo->num_components > MAX_COMPONENTS)
        ERREXIT2(cinfo, JERR_COMPONENT_COUNT, cinfo->num_components,
            MAX_COMPONENTS);
    for (ci = 0; ci < cinfo->num_components; ci++) {
        SET_COMP(ci, ci, 1, 1, 0, 0, 0);
    }
    break;
default:
    ERREXIT(cinfo, JERR_BAD_J_COLORSPACE);
}
}

#ifdef C_PROGRESSIVE_SUPPORTED

LOCAL(jpeg_scan_info *)
fill_a_scan (jpeg_scan_info * scan_ptr, int ci,
             int Ss, int Se, int Ah, int Al)
/* Support routine: generate one scan for specified component */
{
    scan_ptr->comps_in_scan = 1;
    scan_ptr->component_index[0] = ci;
}

```

```

scanptr->Ss = Ss;
scanptr->Se = Se;
scanptr->Ah = Ah;
scanptr->Al = Al;
scanptr++;
return scanptr;
}

LOCAL(jpeg_scan_info *)
fill_scans (jpeg_scan_info * scanptr, int ncomps,
            int Ss, int Se, int Ah, int Al)
/* Support routine: generate one scan for each component */
{
    int ci;

    for (ci = 0; ci < ncomps; ci++) {
        scanptr->comps_in_scan = 1;
        scanptr->component_index[0] = ci;
        scanptr->Ss = Ss;
        scanptr->Se = Se;
        scanptr->Ah = Ah;
        scanptr->Al = Al;
        scanptr++;
    }
    return scanptr;
}

LOCAL(jpeg_scan_info *)
fill_dc_scans (jpeg_scan_info * scanptr, int ncomps, int Ah, int Al)
/* Support routine: generate interleaved DC scan if possible, else N scans */
{
    int ci;

    if (ncomps <= MAX_COMPS_IN_SCAN) {
        /* Single interleaved DC scan */
        scanptr->comps_in_scan = ncomps;
        for (ci = 0; ci < ncomps; ci++)
            scanptr->component_index[ci] = ci;
        scanptr->Ss = scanptr->Se = 0;
        scanptr->Ah = Ah;
        scanptr->Al = Al;
        scanptr++;
    } else {
        /* Noninterleaved DC scan for each component */
        scanptr = fill_scans(scanptr, ncomps, 0, 0, Ah, Al);
    }
    return scanptr;
}

/* Create a recommended progressive-JPEG script.
   cinfo->num_components and cinfo->jpeg_color_space must be correct.
*/

GLOBAL(void)
jpeg_simple_progression (j_compress_ptr cinfo)
{
    int ncomps = cinfo->num_components;
    int nscans;
    jpeg_scan_info * scanptr;

    /* Safety check to ensure start_compress not called yet. */
    if (cinfo->global_state != CSTATE_START)
        ERREXIT1(cinfo, JERR_BAD_STATE, cinfo->global_state);

    /* Figure space needed for script. Calculation must match code below! */
    if (ncomps == 3 && cinfo->jpeg_color_space == JCS_YCbCr) {
        /* Custom script for YCbCr color images. */
        nscans = 10;
    } else {
        /* All-purpose script for other color spaces. */
        if (ncomps > MAX_COMPS_IN_SCAN)
            nscans = 6 * ncomps; /* 2 DC + 4 AC scans per component */
        else
            nscans = 2 + 4 * ncomps; /* 2 DC scans; 4 AC scans per component */
    }

    /* Allocate space for script.
       * We need to put it in the permanent pool in case the application performs
    */

```



```

* multiple compressions without changing the settings. To avoid memory
* leak if jpeg_simple_progression is called repeatedly for the same JPEG
* object, we try to re-use previously allocated space, and we allocate
* enough space to handle YCbCr even if initially asked for grayscale.
*/
if (cinfo->script_space == NULL || cinfo->script_space_size < nscans) {
    cinfo->script_space_size = MAX(nscans, 10);
    cinfo->script_space = (jpeg_scan_info *)
        (*cinfo->mem->alloc_small) ((j_common_ptr) cinfo, JPOOL_PERMANENT,
            cinfo->script_space_size * sizeof(jpeg_scan_info));
}
scanptr = cinfo->script_space;
cinfo->scan_info = scanptr;
cinfo->num_scans = nscans;

if (ncomps == 3 && cinfo->jpeg_color_space == JCS_YCbCr) {
    /* Custom script for YCbCr color images. */
    /* Initial DC scan */
    scanptr = fill_dc_scans(scanptr, ncomps, 0, 1);
    /* Initial AC scan: get some luma data out in a hurry */
    scanptr = fill_a_scan(scanptr, 0, 1, 5, 0, 2);
    /* Chroma data is too small to be worth expending many scans on */
    scanptr = fill_a_scan(scanptr, 2, 1, 63, 0, 1);
    scanptr = fill_a_scan(scanptr, 1, 1, 63, 0, 1);
    /* Complete spectral selection for luma AC */
    scanptr = fill_a_scan(scanptr, 0, 6, 63, 0, 2);
    /* Refine next bit of luma AC */
    scanptr = fill_a_scan(scanptr, 0, 1, 63, 2, 1);
    /* Finish DC successive approximation */
    scanptr = fill_dc_scans(scanptr, ncomps, 1, 0);
    /* Finish AC successive approximation */
    scanptr = fill_a_scan(scanptr, 2, 1, 63, 1, 0);
    scanptr = fill_a_scan(scanptr, 1, 1, 63, 1, 0);
    /* Luma bottom bit comes last since it's usually largest scan */
    scanptr = fill_a_scan(scanptr, 0, 1, 63, 1, 0);
} else {
    /* All-purpose script for other color spaces. */
    /* Successive approximation first pass */
    scanptr = fill_dc_scans(scanptr, ncomps, 0, 1);
    scanptr = fill_scans(scanptr, ncomps, 1, 5, 0, 2);
    scanptr = fill_scans(scanptr, ncomps, 6, 63, 0, 2);
    /* Successive approximation second pass */
    scanptr = fill_scans(scanptr, ncomps, 1, 63, 2, 1);
    /* Successive approximation final pass */
    scanptr = fill_dc_scans(scanptr, ncomps, 1, 0);
    scanptr = fill_scans(scanptr, ncomps, 1, 63, 1, 0);
}
#endif /* C_PROGRESSIVE_SUPPORTED */

```

```

/*
 * jcphuff.c
 *
 * Copyright (C) 1995-1997, Thomas G. Lane.
 * This file is part of the Independent JPEG Group's software.
 * For conditions of distribution and use, see the accompanying README file.
 *
 * This file contains Huffman entropy encoding routines for progressive JPEG.
 *
 * We do not support output suspension in this module, since the library
 * currently does not allow multiple-scan files to be written with output
 * suspension.
 */

#define JPEG_INTERNALS
#include "jinclude.h"
#include "jpeglib.h"
#include "jchuff.h" /* Declarations shared with jchuff.c */

#ifdef C_PROGRESSIVE_SUPPORTED

/* Expanded entropy encoder object for progressive Huffman encoding. */

typedef struct {
  struct jpeg_entropy_encoder pub; /* public fields */

  /* Mode flag: TRUE for optimization, FALSE for actual data output */
  boolean gather_statistics;

  /* Bit-level coding status.
   * next_output_byte/free_in_buffer are local copies of cinfo->dest fields.
   */
  OCTET * next_output_byte; /* => next byte to write in buffer */
  size_t free_in_buffer; /* # of byte spaces remaining in buffer */
  INT32 put_buffer; /* current bit-accumulation buffer */
  int put_bits; /* # of bits now in it */
  j_compress_ptr cinfo; /* link to cinfo (needed for dump_buffer) */

  /* Coding status for DC components */
  int last_dc_val[MAX_COMPS_IN_SCAN]; /* last DC coef for each component */

  /* Coding status for AC components */
  int ac_tbl_no; /* the table number of the single component */
  unsigned int EOBRUN; /* run length of EOBs */
  unsigned int BE; /* # of buffered correction bits before MCU */
  char * bit_buffer; /* buffer for correction bits (1 per char) */
  /* packing correction bits tightly would save some space but cost time... */

  unsigned int restarts_to_go; /* MCUs left in this restart interval */
  int next_restart_num; /* next restart number to write (0-7) */

  /* Pointers to derived tables (these workspaces have image lifespan).
   * Since any one scan codes only DC or only AC, we only need one set
   * of tables, not one for DC and one for AC.
   */
  c_derived_tbl * derived_tbls[NUM_HUFF_TBLS];

  /* Statistics tables for optimization; again, one set is enough */
  long * count_ptrs[NUM_HUFF_TBLS];
} phuff_entropy_encoder;

typedef phuff_entropy_encoder * phuff_entropy_ptr;

/* MAX_CORR_BITS is the number of bits the AC refinement correction-bit
 * buffer can hold. Larger sizes may slightly improve compression, but
 * 1000 is already well into the realm of overkill.
 * The minimum safe size is 64 bits.
 */

#define MAX_CORR_BITS 1000 /* Max # of correction bits I can buffer */

/* IRIGHT_SHIFT is like RIGHT_SHIFT, but works on int rather than INT32.
 * We assume that int right shift is unsigned if INT32 right shift is,
 * which should be safe.
 */

#ifdef RIGHT_SHIFT_IS_UNSIGNED
#define ISHIFT_TEMPS int ishift_temp;
#define IRIGHT_SHIFT(x,shft) \
  ((ishift_temp = (x)) < 0 ? \

```

```

        (ishift_temp >> (shft)) | ((-0) << (16-(shft))) : \
        (ishift_temp >> (shft)))
#else
#define ISHIFT_TEMPS
#define IRIGHT_SHIFT(x,shft)    ((x) >> (shft))
#endif

/* Forward declarations */
METHODDEF(boolean) encode_mcu_DC_first JPP((j_compress_ptr cinfo,
        JBLOCKROW *MCU_data));
METHODDEF(boolean) encode_mcu_AC_first JPP((j_compress_ptr cinfo,
        JBLOCKROW *MCU_data));
METHODDEF(boolean) encode_mcu_DC_refine JPP((j_compress_ptr cinfo,
        JBLOCKROW *MCU_data));
METHODDEF(boolean) encode_mcu_AC_refine JPP((j_compress_ptr cinfo,
        JBLOCKROW *MCU_data));
METHODDEF(void) finish_pass_phuff JPP((j_compress_ptr cinfo));
METHODDEF(void) finish_pass_gather_phuff JPP((j_compress_ptr cinfo));

/*
 * Initialize for a Huffman-compressed scan using progressive JPEG.
 */

METHODDEF(void)
start_pass_phuff (j_compress_ptr cinfo, boolean gather_statistics)
{
    phuff_entropy_ptr entropy = (phuff_entropy_ptr) cinfo->entropy;
    boolean is_DC_band;
    int ci, tbl;
    jpeg_component_info * compptr;

    entropy->cinfo = cinfo;
    entropy->gather_statistics = gather_statistics;

    is_DC_band = (cinfo->Ss == 0);

    /* We assume jcmaster.c already validated the scan parameters. */

    /* Select execution routines */
    if (cinfo->Ah == 0) {
        if (is_DC_band)
            entropy->pub.encode_mcu = encode_mcu_DC_first;
        else
            entropy->pub.encode_mcu = encode_mcu_AC_first;
    } else {
        if (is_DC_band)
            entropy->pub.encode_mcu = encode_mcu_DC_refine;
        else {
            entropy->pub.encode_mcu = encode_mcu_AC_refine;
            /* AC refinement needs a correction bit buffer */
            if (entropy->bit_buffer == NULL)
                entropy->bit_buffer = (char *)
                    (*cinfo->mem->alloc_small) ((j_common_ptr) cinfo, JPOOL_IMAGE,
                        MAX_CORR_BITS * SIZEOF(char));
        }
    }

    if (gather_statistics)
        entropy->pub.finish_pass = finish_pass_gather_phuff;
    else
        entropy->pub.finish_pass = finish_pass_phuff;

    /* Only DC coefficients may be interleaved, so cinfo->comps_in_scan = 1
     * for AC coefficients.
     */
    for (ci = 0; ci < cinfo->comps_in_scan; ci++) {
        compptr = cinfo->cur_comp_info[ci];
        /* Initialize DC predictions to 0 */
        entropy->last_dc_val[ci] = 0;
        /* Get table index */
        if (is_DC_band) {
            if (cinfo->Ah != 0) /* DC refinement needs no table */
                continue;
            tbl = compptr->dc_tbl_no;
        } else {
            entropy->ac_tbl_no = tbl = compptr->ac_tbl_no;
        }
        if (gather_statistics) {
            /* Check for invalid table index */
            /* (make_c_derived_tbl does this in the other path) */

```

```

    if (tbl < 0 || tbl >= NUM_HUFF_TBLS)
        ERREXIT1(cinfo, JERR_HUFF_TABLE, tbl);
    /* Allocate and zero the statistics tables */
    /* Note that jpeg_gen_optimal_table expects 257 entries in each table! */
    if (entropy->count_ptrs[tbl] == NULL)
        entropy->count_ptrs[tbl] = (long *)
            (*cinfo->mem->alloc_small) ((j_common_ptr) cinfo, JPOOL_IMAGE,
                257 * sizeof(long));
    MEMZERO(entropy->count_ptrs[tbl], 257 * sizeof(long));
} else {
    /* Compute derived values for Huffman table */
    /* We may do this more than once for a table, but it's not expensive */
    jpeg_make_c_derived_tbl(cinfo, is_DC_band, tbl,
        & entropy->derived_tbls[tbl]);
}
}

/* Initialize AC stuff */
entropy->EOBRUN = 0;
entropy->BE = 0;

/* Initialize bit buffer to empty */
entropy->put_buffer = 0;
entropy->put_bits = 0;

/* Initialize restart stuff */
entropy->restarts_to_go = cinfo->restart_interval;
entropy->next_restart_num = 0;
}

/* Outputting bytes to the file.
 * NB: these must be called only when actually outputting,
 * that is, entropy->gather_statistics == FALSE.
 */

/* Emit a byte */
#define emit_byte(entropy, val) \
    { *(entropy)->next_output_byte++ = (JOCTET) (val); \
      if (--(entropy)->free_in_buffer == 0) \
          dump_buffer(entropy); }

LOCAL(void)
dump_buffer (phuff_entropy_ptr entropy)
/* Empty the output buffer; we do not support suspension in this module. */
{
    struct jpeg_destination_mgr * dest = entropy->cinfo->dest;

    if (! (*dest->empty_output_buffer) (entropy->cinfo))
        ERREXIT(entropy->cinfo, JERR_CANT_SUSPEND);
    /* After a successful buffer dump, must reset buffer pointers */
    entropy->next_output_byte = dest->next_output_byte;
    entropy->free_in_buffer = dest->free_in_buffer;
}

/* Outputting bits to the file */

/* Only the right 24 bits of put_buffer are used; the valid bits are
 * left-justified in this part. At most 16 bits can be passed to emit_bits
 * in one call, and we never retain more than 7 bits in put_buffer
 * between calls, so 24 bits are sufficient.
 */

INLINE
LOCAL(void)
emit_bits (phuff_entropy_ptr entropy, unsigned int code, int size)
/* Emit some bits, unless we are in gather mode */
{
    /* This routine is heavily used, so it's worth coding tightly. */
    register INT32 put_buffer = (INT32) code;
    register int put_bits = entropy->put_bits;

    /* if size is 0, caller used an invalid Huffman table entry */
    if (size == 0)
        ERREXIT(entropy->cinfo, JERR_HUFF_MISSING_CODE);

    if (entropy->gather_statistics)
        return;
    /* do nothing if we're only getting stats */
}

```

```

put_buffer &= (((INT32) 1) << e) - 1; /* mask off any extra bits in code */
put_bits += size; /* new number of bits in buffer */
put_buffer <<= 24 - put_bits; /* align incoming bits */
put_buffer |= entropy->put_buffer; /* and merge with old buffer contents */

while (put_bits >= 8) {
    int c = (int) ((put_buffer >> 16) & 0xFF);

    emit_byte(entropy, c);
    if (c == 0xFF) { /* need to stuff a zero byte? */
        emit_byte(entropy, 0);
    }
    put_buffer <<= 8;
    put_bits -= 8;
}

entropy->put_buffer = put_buffer; /* update variables */
entropy->put_bits = put_bits;
}

```

```

LOCAL(void)
flush_bits (phuff_entropy_ptr entropy)
{
    emit_bits(entropy, 0x7F, 7); /* fill any partial byte with ones */
    entropy->put_buffer = 0; /* and reset bit-buffer to empty */
    entropy->put_bits = 0;
}

```

```

/* Emit (or just count) a Huffman symbol.

```

```

INLINE
LOCAL(void)
emit_symbol (phuff_entropy_ptr entropy, int tbl_no, int symbol)
{
    if (entropy->gather_statistics)
        entropy->count_ptrs[tbl_no][symbol]++;
    else {
        c_derived_tbl * tbl = entropy->derived_tbls[tbl_no];
        emit_bits(entropy, tbl->ehufco[symbol], tbl->ehufsi[symbol]);
    }
}

```

```

/* Emit bits from a correction bit buffer.
*/

```

```

LOCAL(void)
emit_buffered_bits (phuff_entropy_ptr entropy, char * bufstart,
                    unsigned int nbits)
{
    if (entropy->gather_statistics)
        return; /* no real work */

    while (nbits > 0) {
        emit_bits(entropy, (unsigned int) (*bufstart), 1);
        bufstart++;
        nbits--;
    }
}

```

```

/*
 * Emit any pending EOBRUN symbol.
*/

```

```

LOCAL(void)
emit_eobrun (phuff_entropy_ptr entropy)
{
    register int temp, nbits;

    if (entropy->EOBRUN > 0) { /* if there is any pending EOBRUN */

```

```

temp = entropy->EOBRUN;
nbits = 0;
while ((temp >= 1))
    nbits++;
/* safety check: shouldn't happen given limited correction-bit buffer */
if (nbits > 14)
    ERREXIT(entropy->cinfo, JERR_HUFF_MISSING_CODE);

emit_symbol(entropy, entropy->ac_tbl_no, nbits << 4);
if (nbits)
    emit_bits(entropy, entropy->EOBRUN, nbits);

entropy->EOBRUN = 0;

/* Emit any buffered correction bits */
emit_buffered_bits(entropy, entropy->bit_buffer, entropy->BE);
entropy->BE = 0;
}
}

```

```

/*
 * Emit a restart marker & resynchronize predictions.
 */

```

```

LOCAL(void)
emit_restart (phuff_entropy_ptr entropy, int restart_num)
{
    int ci;

    emit_eobrun(entropy);

    if (! entropy->gather_statistics) {
        flush_bits(entropy);
        emit_byte(entropy, 0xFF);
        emit_byte(entropy, JPEG_RST0 + restart_num);
    }

    if (entropy->cinfo->Ss == 0) {
        /* Re-initialize DC predictions to 0 */
        for (ci = 0; ci < entropy->cinfo->comps_in_scan; ci++)
            entropy->last_dc_val[ci] = 0;
    } else {
        /* Re-initialize all AC-related fields to 0 */
        entropy->EOBRUN = 0;
        entropy->BE = 0;
    }
}

```

```

/* MCU encoding for DC initial scan (either spectral selection,
 * or first pass of successive approximation).
 */

```

```

METHODDEF(boolean)
encode_mcu_DC_first (j_compress_ptr cinfo, JBLOCKROW *MCU_data)
{
    phuff_entropy_ptr entropy = (phuff_entropy_ptr) cinfo->entropy;
    register int temp, temp2;
    register int nbits;
    int blk, ci;
    int Al = cinfo->Al;
    JBLOCKROW block;
    jpeg_component_info * comp_ptr;
    ISHIFT_TEMPS

    entropy->next_output_byte = cinfo->dest->next_output_byte;
    entropy->free_in_buffer = cinfo->dest->free_in_buffer;

    /* Emit restart marker if needed */
    if (cinfo->restart_interval)
        if (entropy->restarts_to_go == 0)
            emit_restart(entropy, entropy->next_restart_num);

    /* Encode the MCU data blocks */
    for (blk = 0; blk < cinfo->blocks_in_MCU; blk++) {
        block = MCU_data[blk];
        ci = cinfo->MCU_membership[blk];
        comp_ptr = cinfo->cur_comp_info[ci];
    }
}

```

```

/* Compute the DC value and the required point transform by
 * This is simply an arithmetic right shift.
 */
temp2 = IRIGHT_SHIFT((int) ((*block)[0]), A1);

/* DC differences are figured on the point-transformed values. */
temp = temp2 - entropy->last_dc_val[ci];
entropy->last_dc_val[ci] = temp2;

/* Encode the DC coefficient difference per section G.1.2.1 */
temp2 = temp;
if (temp < 0) {
    temp = -temp; /* temp is abs value of input */
    /* For a negative input, want temp2 = bitwise complement of abs(input) */
    /* This code assumes we are on a two's complement machine */
    temp2--;
}

/* Find the number of bits needed for the magnitude of the coefficient */
nbits = 0;
while (temp) {
    nbits++;
    temp >>= 1;
}
/* Check for out-of-range coefficient values.
 * Since we're encoding a difference, the range limit is twice as much.
 */
if (nbits > MAX_COEF_BITS+1)
    ERREXIT(cinfo, JERR_BAD_DCT_COEF);

/* Count/emit the Huffman-coded symbol for the number of bits */
emit_symbol(entropy, compptr->dc_tbl_no, nbits);

/* Emit that number of bits of the value, if positive, */
/* or the complement of its magnitude, if negative. */
if (nbits) /* emit_bits rejects calls with size 0 */
    emit_bits(entropy, (unsigned int) temp2, nbits);

cinfo->dest->next_output_byte = entropy->next_output_byte;
cinfo->dest->free_in_buffer = entropy->free_in_buffer;

/* Update restart-interval state too */
if (cinfo->restart_interval) {
    if (entropy->restarts_to_go == 0) {
        entropy->restarts_to_go = cinfo->restart_interval;
        entropy->next_restart_num++;
        entropy->next_restart_num &= 7;
    }
    entropy->restarts_to_go--;
}

return TRUE;
}

/*
 * MCU encoding for AC initial scan (either spectral selection,
 * or first pass of successive approximation).
 */

METHODDEF(boolean)
encode_mcu_AC_first (j_compress_ptr cinfo, JBLOCKROW *MCU_data)
{
    phuff_entropy_ptr entropy = (phuff_entropy_ptr) cinfo->entropy;
    register int temp, temp2;
    register int nbits;
    register int r, k;
    int Se = cinfo->Se;
    int A1 = cinfo->A1;
    JBLOCKROW block;

    entropy->next_output_byte = cinfo->dest->next_output_byte;
    entropy->free_in_buffer = cinfo->dest->free_in_buffer;

    /* Emit restart marker if needed */
    if (cinfo->restart_interval)
        if (entropy->restarts_to_go == 0)
            emit_restart(entropy, entropy->next_restart_num);

```

```

)

/*
 * MCU encoding for DC successive approximation refinement scan.
 * Note: we assume such scans can be multi-component, although the spec
 * is not very clear on the point.
 */

```

```

METHODDEF(boolean)
encode_mcu_DC_refine (j_compress_ptr cinfo, JBLOCKROW *MCU_data)
{
    phuff_entropy_ptr entropy = (phuff_entropy_ptr) cinfo->entropy;
    register int temp;
    int blkn;
    int A1 = cinfo->A1;
    JBLOCKROW block;

    entropy->next_output_byte = cinfo->dest->next_output_byte;
    entropy->free_in_buffer = cinfo->dest->free_in_buffer;

    /* Emit restart marker if needed */
    if (cinfo->restart_interval)
        if (entropy->restarts_to_go == 0)
            emit_restart(entropy, entropy->next_restart_num);

    /* Encode the MCU data blocks */
    for (blkn = 0; blkn < cinfo->blocks_in_MCU; blkn++) {
        block = MCU_data[blkn];

        /* We simply emit the A1'th bit of the DC coefficient value. */
        temp = (*block)[0];
        emit_bits(entropy, (unsigned int) (temp >> A1), 1);

        cinfo->dest->next_output_byte = entropy->next_output_byte;
        cinfo->dest->free_in_buffer = entropy->free_in_buffer;

        /* Update restart-interval state too */
        if (cinfo->restart_interval) {
            if (entropy->restarts_to_go == 0) {
                entropy->restarts_to_go = cinfo->restart_interval;
                entropy->next_restart_num++;
                entropy->next_restart_num &= 7;
            }
            entropy->restarts_to_go--;
        }

        return TRUE;
    }
}
/*
 * MCU encoding for AC successive approximation refinement scan.
 */

```

```

METHODDEF(boolean)
encode_mcu_AC_refine (j_compress_ptr cinfo, JBLOCKROW *MCU_data)
{
    phuff_entropy_ptr entropy = (phuff_entropy_ptr) cinfo->entropy;
    register int temp;
    register int r, k;
    int EOB;
    char *BR_buffer;
    unsigned int BR;
    int Se = cinfo->Se;
    int A1 = cinfo->A1;
    JBLOCKROW block;
    int absvalues[DCTSIZE2];

    entropy->next_output_byte = cinfo->dest->next_output_byte;
    entropy->free_in_buffer = cinfo->dest->free_in_buffer;

    /* Emit restart marker if needed */
    if (cinfo->restart_interval)
        if (entropy->restarts_to_go == 0)
            emit_restart(entropy, entropy->next_restart_num);

    /* Encode the MCU data block */
    block = MCU_data[0];

```



```

/* Encode the MCU data block
block = MCU_data[0];

/* Encode the AC coefficients per section G.1.2.2, fig. G.3 */
r = 0;          /* r = run length of zeros */

for (k = cinfo->Ss; k <= Se; k++) {
    if ((temp = (*block)[jpeg_natural_order[k]]) == 0) {
        r++;
        continue;
    }
    /* We must apply the point transform by A1. For AC coefficients this
     * is an integer division with rounding towards 0. To do this portably
     * in C, we shift after obtaining the absolute value; so the code is
     * interwoven with finding the abs value (temp) and output bits (temp2).
     */
    if (temp < 0) {
        temp = -temp;          /* temp is abs value of input */
        temp >>= A1;          /* apply the point transform */
        /* For a negative coef, want temp2 = bitwise complement of abs(coef) */
        temp2 = ~temp;
    } else {
        temp >>= A1;          /* apply the point transform */
        temp2 = temp;
    }
    /* Watch out for case that nonzero coef is zero after point transform */
    if (temp == 0) {
        r++;
        continue;
    }
    /* Emit any pending EOBRUN */
    if (entropy->EOBRUN > 0)
        emit_eobrun(entropy);
    /* if run length > 15, must emit special run-length-16 codes (0xF0) */
    while (r > 15) {
        emit_symbol(entropy, entropy->ac_tbl_no, 0xF0);
        r -= 16;
    }
    /* Find the number of bits needed for the magnitude of the coefficient */
    nbits = 1;          /* there must be at least one 1 bit */
    while ((temp >>= 1))
        nbits++;
    /* Check for out-of-range coefficient values */
    if (nbits > MAX_COEF_BITS)
        ERREXIT(cinfo, JERR_BAD_DCT_COEF);
    /* Count/emit Huffman symbol for run length / number of bits */
    emit_symbol(entropy, entropy->ac_tbl_no, (r < 4) + nbits);
    /* Emit that number of bits of the value, if positive, */
    /* or the complement of its magnitude, if negative. */
    emit_bits(entropy, (unsigned int) temp2, nbits);

    r = 0;          /* reset zero run length */
}

if (r > 0) {          /* If there are trailing zeroes, */
    entropy->EOBRUN++; /* count an EOB */
    if (entropy->EOBRUN == 0x7FFF)
        emit_eobrun(entropy); /* force it out to avoid overflow */
}

cinfo->dest->next_output_byte = entropy->next_output_byte;
cinfo->dest->free_in_buffer = entropy->free_in_buffer;

/* Update restart-interval state too */
if (cinfo->restart_interval) {
    if (entropy->restarts_to_go == 0) {
        entropy->restarts_to_go = cinfo->restart_interval;
        entropy->next_restart_num++;
        entropy->next_restart_num &= 7;
    }
    entropy->restarts_to_go--;
}

return TRUE;

```

```

/* It is convenient to make a pre-pass to determine the transform
 * coefficients' absolute values and the EOB position.
 */
EOB = 0;
for (k = cinfo->Ss; k <= Se; k++) {
    temp = (*block)[jpeg_natural_order[k]];
    /* We must apply the point transform by A1. For AC coefficients this
     * is an integer division with rounding towards 0. To do this portably
     * in C, we shift after obtaining the absolute value.
     */
    if (temp < 0)
        temp = -temp;      /* temp is abs value of input */
    temp >>= A1;           /* apply the point transform */
    absvalues[k] = temp;    /* save abs value for main pass */
    if (temp == 1)
        EOB = k;          /* EOB = index of last newly-nonzero coef */
}

/* Encode the AC coefficients per section G.1.2.3, fig. G.7 */

r = 0;                  /* r = run length of zeros */
BR = 0;                 /* BR = count of buffered bits added now */
BR_buffer = entropy->bit_buffer + entropy->BE; /* Append bits to buffer */

for (k = cinfo->Ss; k <= Se; k++) {
    if ((temp = absvalues[k]) == 0) {
        r++;
        continue;
    }

    /* Emit any required ZRLs, but not if they can be folded into EOB */
    while (r > 15 && k <= EOB) {
        /* emit any pending EOBRUN and the BE correction bits */
        emit_eobrun(entropy);
        /* Emit ZRL */
        emit_symbol(entropy, entropy->ac_tbl_no, 0xF0);
        r -= 16;
        /* Emit buffered correction bits that must be associated with ZRL */
        emit_buffered_bits(entropy, BR_buffer, BR);
        BR_buffer = entropy->bit_buffer; /* BE bits are gone now */
        BR = 0;
    }

    /* If the coef was previously nonzero, it only needs a correction bit.
     * NOTE: a straight translation of the spec's figure G.7 would suggest
     * that we also need to test r > 15. But if r > 15, we can only get here
     * if k > EOB, which implies that this coefficient is not 1.
     */
    if (temp > 1) {
        /* The correction bit is the next bit of the absolute value. */
        BR_buffer[BR++] = (char) (temp & 1);
        continue;
    }

    /* Emit any pending EOBRUN and the BE correction bits */
    emit_eobrun(entropy);

    /* Count/emit Huffman symbol for run length / number of bits */
    emit_symbol(entropy, entropy->ac_tbl_no, (r << 4) + 1);

    /* Emit output bit for newly-nonzero coef */
    temp = ((*block)[jpeg_natural_order[k]] < 0) ? 0 : 1;
    emit_bits(entropy, (unsigned int) temp, 1);

    /* Emit buffered correction bits that must be associated with this code */
    emit_buffered_bits(entropy, BR_buffer, BR);
    BR_buffer = entropy->bit_buffer; /* BE bits are gone now */
    BR = 0;
    r = 0;          /* reset zero run length */
}

if (r > 0 || BR > 0) { /* If there are trailing zeroes, */
    entropy->EOBRUN++;   /* count an EOB */
    entropy->BE += BR;   /* concat my correction bits to older ones */
    /* We force out the EOB if we risk either:
     * 1. overflow of the EOB counter;
     * 2. overflow of the correction bit buffer during the next MCU.
     */
    if (entropy->EOBRUN == 0x7FFF || entropy->BE > (MAX_CORR_BITS-DCTSIZE2+1))

```

```

    emit_eobrun(entropy);
}

cinfo->dest->next_output_byte = entropy->next_output_byte;
cinfo->dest->free_in_buffer = entropy->free_in_buffer;

/* Update restart-interval state too */
if (cinfo->restart_interval) {
    if (entropy->restarts_to_go == 0) {
        entropy->restarts_to_go = cinfo->restart_interval;
        entropy->next_restart_num++;
        entropy->next_restart_num &= 7;
    }
    entropy->restarts_to_go--;
}

return TRUE;
}

/*
 * Finish up at the end of a Huffman-compressed progressive scan.
 */

METHODDEF(void)
finish_pass_phuff (j_compress_ptr cinfo)
{
    phuff_entropy_ptr entropy = (phuff_entropy_ptr) cinfo->entropy;

    entropy->next_output_byte = cinfo->dest->next_output_byte;
    entropy->free_in_buffer = cinfo->dest->free_in_buffer;

    /* Flush out any buffered data */
    emit_eobrun(entropy);
    flush_bits(entropy);

    cinfo->dest->next_output_byte = entropy->next_output_byte;
    cinfo->dest->free_in_buffer = entropy->free_in_buffer;
}

/*
 * Finish up a statistics-gathering pass and create the new Huffman tables.
 */

METHODDEF(void)
finish_pass_gather_phuff (j_compress_ptr cinfo)
{
    phuff_entropy_ptr entropy = (phuff_entropy_ptr) cinfo->entropy;
    boolean is_DC_band;
    int ci, tbl;
    jpeg_component_info * compptr;
    JHUFF_TBL **htblptr;
    boolean did[NUM_HUFF_TBLS];

    /* Flush out buffered data (all we care about is counting the EOB symbol) */
    emit_eobrun(entropy);

    is_DC_band = (cinfo->Ss == 0);

    /* It's important not to apply jpeg_gen_optimal_table more than once
     * per table, because it clobbers the input frequency counts!
     */
    MEMZERO(did, sizeof(did));

    for (ci = 0; ci < cinfo->comps_in_scan; ci++) {
        compptr = cinfo->cur_comp_info[ci];
        if (is_DC_band) {
            if (cinfo->Ah != 0) /* DC refinement needs no table */
                continue;
            tbl = compptr->dc_tbl_no;
        } else {
            tbl = compptr->ac_tbl_no;
        }
        if (! did[tbl]) {
            if (is_DC_band)
                htblptr = & cinfo->dc_huff_tbl_ptrs[tbl];
            else
                htblptr = & cinfo->ac_huff_tbl_ptrs[tbl];
            if (*htblptr == NULL)

```

```

        *htblptr = jpeg_alloc_huff_table((j_common_ptr) cinfo);
        jpeg_gen_optimal_table(cinfo, *htblptr, entropy->count_ptrs);
        did[tbl] = TRUE;
    }
}

/*
 * Module initialization routine for progressive Huffman entropy encoding.
 */

GLOBAL(void)
jinit_phuff_encoder (j_compress_ptr cinfo)
{
    phuff_entropy_ptr entropy;
    int i;

    entropy = (phuff_entropy_ptr)
        (*cinfo->mem->alloc_small) ((j_common_ptr) cinfo, JPOOL_IMAGE,
            sizeof(phuff_entropy_encoder));
    cinfo->entropy = (struct jpeg_entropy_encoder *) entropy;
    entropy->pub.start_pass = start_pass_phuff;

    /* Mark tables unallocated */
    for (i = 0; i < NUM_HUFF_TBLS; i++) {
        entropy->derived_tbls[i] = NULL;
        entropy->count_ptrs[i] = NULL;
    }
    entropy->bit_buffer = NULL; /* needed only in AC refinement scan */
}
#endif /* C_PROGRESSIVE_SUPPORTED */

```

```

/*
 * jcprepct.c
 *
 * Copyright (C) 1994-1996, Thomas G. Lane.
 * This file is part of the Independent JPEG Group's software.
 * For conditions of distribution and use, see the accompanying README file.
 *
 * This file contains the compression preprocessing controller.
 * This controller manages the color conversion, downsampling,
 * and edge expansion steps.
 *
 * Most of the complexity here is associated with buffering input rows
 * as required by the downsampler. See the comments at the head of
 * jcsample.c for the downsampler's needs.
 */

#define JPEG_INTERNALS
#include "jinclude.h"
#include "jpeglib.h"

/* At present, jcsample.c can request context rows only for smoothing.
 * In the future, we might also need context rows for CCIR601 sampling
 * or other more-complex downsampling procedures. The code to support
 * context rows should be compiled only if needed.
 */
#ifdef INPUT_SMOOTHING_SUPPORTED
#define CONTEXT_ROWS_SUPPORTED
#endif

/*
 * For the simple (no-context-row) case, we just need to buffer one
 * row group's worth of pixels for the downsampling step. At the bottom of
 * the image, we pad to a full row group by replicating the last pixel row.
 * The downsampler's last output row is then replicated if needed to pad
 * out to a full iMCU row.
 *
 * When providing context rows, we must buffer three row groups' worth of
 * pixels. Three row groups are physically allocated, but the row pointer
 * arrays are made five row groups high, with the extra pointers above and
 * below "wrapping around" to point to the last and first real row groups.
 * This allows the downsampler to access the proper context rows.
 * At the top and bottom of the image, we create dummy context rows by
 * copying the first or last real pixel row. This copying could be avoided
 * by pointer hacking as is done in jdmainct.c, but it doesn't seem worth the
 * trouble on the compression side.
 */

/* Private buffer controller object */
typedef struct {
  struct jpeg_c_prep_controller pub; /* public fields */

  /* Downsampling input buffer. This buffer holds color-converted data
   * until we have enough to do a downsample step.
   */
  JSAMPARRAY color_buf[MAX_COMPONENTS];

  JDIMENSION rows_to_go; /* counts rows remaining in source image */
  int next_buf_row; /* index of next row to store in color_buf */

#ifdef CONTEXT_ROWS_SUPPORTED /* only needed for context case */
  int this_row_group; /* starting row index of group to process */
  int next_buf_stop; /* downsample when we reach this index */
#endif
} my_prep_controller;

typedef my_prep_controller * my_prep_ptr;

/*
 * Initialize for a processing pass.
 */
METHODDEF(void)
start_pass_prep (j_compress_ptr cinfo, J_BUF_MODE pass_mode)
{
  my_prep_ptr prep = (my_prep_ptr) cinfo->prep;

```

```

if (pass_mode != JBUF_PASS_
    ERREXIT(cinfo, JERR_BAD_BUFFER_MODE);

/* Initialize total-height counter for detecting bottom of image */
prep->rows_to_go = cinfo->image_height;
/* Mark the conversion buffer empty */
prep->next_buf_row = 0;
#ifdef CONTEXT_ROWS_SUPPORTED
/* Preset additional state variables for context mode.
 * These aren't used in non-context mode, so we needn't test which mode.
 */
prep->this_row_group = 0;
/* Set next_buf_stop to stop after two row groups have been read in. */
prep->next_buf_stop = 2 * cinfo->max_v_samp_factor;
#endif
}

/*
 * Expand an image vertically from height input_rows to height output_rows,
 * by duplicating the bottom row.
 */

LOCAL(void)
expand_bottom_edge (JSAMPARRAY image_data, JDIMENSION num_cols,
    int input_rows, int output_rows)
{
    register int row;

    for (row = input_rows; row < output_rows; row++) {
        jcopy_sample_rows(image_data, input_rows-1, image_data, row,
            1, num_cols);
    }
}

/*
 * Process some data in the simple no-context case.
 *
 * Preprocessor output data is counted in "row groups". A row group
 * is defined to be v_samp_factor sample rows of each component.
 * Downsampling will produce this much data from each max_v_samp_factor
 * input rows.
 */

METHODDEF(void)
pre_process_data (j_compress_ptr cinfo,
    JSAMPARRAY input_buf, JDIMENSION *in_row_ctr,
    JDIMENSION in_rows_avail,
    JSAMPIMAGE output_buf, JDIMENSION *out_row_group_ctr,
    JDIMENSION out_row_groups_avail)
{
    my_prep_ptr prep = (my_prep_ptr) cinfo->prep;
    int numrows, ci;
    JDIMENSION inrows;
    jpeg_component_info * comp_ptr;

    while (*in_row_ctr < in_rows_avail &&
        *out_row_group_ctr < out_row_groups_avail) {
        /* Do color conversion to fill the conversion buffer. */
        inrows = in_rows_avail - *in_row_ctr;
        numrows = cinfo->max_v_samp_factor - prep->next_buf_row;
        numrows = (int) MIN((JDIMENSION) numrows, inrows);
        (*cinfo->cconvert->color_convert) (cinfo, input_buf + *in_row_ctr,
            prep->color_buf,
            (JDIMENSION) prep->next_buf_row,
            numrows);
        *in_row_ctr += numrows;
        prep->next_buf_row += numrows;
        prep->rows_to_go -= numrows;
        /* If at bottom of image, pad to fill the conversion buffer. */
        if (prep->rows_to_go == 0 &&
            prep->next_buf_row < cinfo->max_v_samp_factor) {
            for (ci = 0; ci < cinfo->num_components; ci++) {
                expand_bottom_edge(prep->color_buf[ci], cinfo->image_width,
                    prep->next_buf_row, cinfo->max_v_samp_factor);
            }
            prep->next_buf_row = cinfo->max_v_samp_factor;
        }
    }
}

```

```

/* If we've filled the conversion buffer, empty it. */
if (prep->next_buf_row == cinfo->max_v_samp_factor) {
    (*cinfo->downsample->downsample) (cinfo,
        prep->color_buf, (JDIMENSION) 0,
        output_buf, *out_row_group_ctr);
    prep->next_buf_row = 0;
    (*out_row_group_ctr)++;
}
/* If at bottom of image, pad the output to a full iMCU height.
 * Note we assume the caller is providing a one-iMCU-height output buffer!
 */
if (prep->rows_to_go == 0 &&
    *out_row_group_ctr < out_row_groups_avail) {
    for (ci = 0, compptr = cinfo->comp_info; ci < cinfo->num_components;
        ci++, compptr++) {
        expand_bottom_edge(output_buf[ci],
            compptr->width_in_blocks * DCTSIZE,
            (int) (*out_row_group_ctr * compptr->v_samp_factor),
            (int) (out_row_groups_avail * compptr->v_samp_factor));
    }
    *out_row_group_ctr = out_row_groups_avail;
    break;
    /* can exit outer loop without test */
}
}
}

#ifdef CONTEXT_ROWS_SUPPORTED

/*
 * Process some data in the context case.
 */
METHODDEF(void)
pre_process_context (j_compress_ptr cinfo,
    JSAMPARRAY input_buf, JDIMENSION *in_row_ctr,
    JDIMENSION in_rows_avail,
    JSAMPIMAGE output_buf, JDIMENSION *out_row_group_ctr,
    JDIMENSION out_row_groups_avail)
{
    my_prep_ptr prep = (my_prep_ptr) cinfo->prep;
    int numrows, ci;
    int buf_height = cinfo->max_v_samp_factor * 3;
    = JDIMENSION inrows;

    while (*out_row_group_ctr < out_row_groups_avail) {
        if (*in_row_ctr < in_rows_avail) {
            /* Do color conversion to fill the conversion buffer. */
            inrows = in_rows_avail - *in_row_ctr;
            numrows = prep->next_buf_stop - prep->next_buf_row;
            numrows = (int) MIN((JDIMENSION) numrows, inrows);
            (*cinfo->cconvert->color_convert) (cinfo, input_buf + *in_row_ctr,
                prep->color_buf,
                (JDIMENSION) prep->next_buf_row,
                numrows);
            /* Pad at top of image, if first time through */
            if (prep->rows_to_go == cinfo->image_height) {
                for (ci = 0; ci < cinfo->num_components; ci++) {
                    int row;
                    for (row = 1; row <= cinfo->max_v_samp_factor; row++) {
                        jcopy_sample_rows(prep->color_buf[ci], 0,
                            prep->color_buf[ci], -row,
                            1, cinfo->image_width);
                    }
                }
                *in_row_ctr += numrows;
                prep->next_buf_row += numrows;
                prep->rows_to_go -= numrows;
            } else {
                /* Return for more data, unless we are at the bottom of the image. */
                if (prep->rows_to_go != 0)
                    break;
                /* When at bottom of image, pad to fill the conversion buffer. */
                if (prep->next_buf_row < prep->next_buf_stop) {
                    for (ci = 0; ci < cinfo->num_components; ci++) {
                        expand_bottom_edge(prep->color_buf[ci], cinfo->image_width,
                            prep->next_buf_row, prep->next_buf_stop);
                    }
                }
                prep->next_buf_row = prep->next_buf_stop;
            }
        }
        (*out_row_group_ctr)++;
    }
}

```

```

    }
}
/* If we've gotten enough data, downsample a row group. */
if (prep->next_buf_row == prep->next_buf_stop) {
    (*cinfo->downsample->downsample) (cinfo,
        prep->color_buf,
        (JDIMENSION) prep->this_row_group,
        output_buf, *out_row_group_ctr);
    (*out_row_group_ctr)++;
    /* Advance pointers with wraparound as necessary. */
    prep->this_row_group += cinfo->max_v_samp_factor;
    if (prep->this_row_group >= buf_height)
        prep->this_row_group = 0;
    if (prep->next_buf_row >= buf_height)
        prep->next_buf_row = 0;
    prep->next_buf_stop = prep->next_buf_row + cinfo->max_v_samp_factor;
}
}
}

/*
 * Create the wrapped-around downsampling input buffer needed for context mode.
 */

LOCAL(void)
create_context_buffer (j_compress_ptr cinfo)
{
    my_prep_ptr prep = (my_prep_ptr) cinfo->prep;
    int rgroup_height = cinfo->max_v_samp_factor;
    int ci, i;
    jpeg_component_info * compptr;
    JSAMPARRAY true_buffer, fake_buffer;

    /* Grab enough space for fake row pointers for all the components;
     * we need five row groups' worth of pointers for each component.
     */
    fake_buffer = (JSAMPARRAY)
        (*cinfo->mem->alloc_small) ((j_common_ptr) cinfo, JPOOL_IMAGE,
            (cinfo->num_components * 5 * rgroup_height) *
            SIZEOF(JSAMPROW));

    for (ci = 0, compptr = cinfo->comp_info; ci < cinfo->num_components;
        ci++, compptr++) {
        /* Allocate the actual buffer space (3 row groups) for this component.
         * We make the buffer wide enough to allow the downsampler to edge-expand
         * horizontally within the buffer, if it so chooses.
         */
        true_buffer = (*cinfo->mem->alloc_sarray)
            ((j_common_ptr) cinfo, JPOOL_IMAGE,
            (JDIMENSION) (((long) compptr->width_in_blocks * DCTSIZE *
                cinfo->max_h_samp_factor) / compptr->h_samp_factor),
            (JDIMENSION) (3 * rgroup_height));
        /* Copy true buffer row pointers into the middle of the fake row array */
        MEMCOPY(fake_buffer + rgroup_height, true_buffer,
            3 * rgroup_height * SIZEOF(JSAMPROW));
        /* Fill in the above and below wraparound pointers */
        for (i = 0; i < rgroup_height; i++) {
            fake_buffer[i] = true_buffer[2 * rgroup_height + i];
            fake_buffer[4 * rgroup_height + i] = true_buffer[i];
        }
        prep->color_buf[ci] = fake_buffer + rgroup_height;
        fake_buffer += 5 * rgroup_height; /* point to space for next component */
    }
}

#endif /* CONTEXT_ROWS_SUPPORTED */

/*
 * Initialize preprocessing controller.
 */

GLOBAL(void)
jinit_c_prep_controller (j_compress_ptr cinfo, boolean need_full_buffer)
{
    my_prep_ptr prep;
    int ci;
    jpeg_component_info * compptr;

```



```

if (need_full_buffer) /* safety check */
    ERREXIT(cinfo, JERR_BAD_BUFFER_MODE);

prep = (my_prep_ptr)
    (*cinfo->mem->alloc_small) ((j_common_ptr) cinfo, JPOOL_IMAGE,
        sizeof(my_prep_controller));
cinfo->prep = (struct jpeg_c_prep_controller *) prep;
prep->pub.start_pass = start_pass_prep;

/* Allocate the color conversion buffer.
 * We make the buffer wide enough to allow the downsampler to edge-expand
 * horizontally within the buffer, if it so chooses.
 */
if (cinfo->downsample->need_context_rows) {
    /* Set up to provide context rows */
#ifdef CONTEXT_ROWS_SUPPORTED
    prep->pub.pre_process_data = pre_process_context;
    create_context_buffer(cinfo);
#else
    ERREXIT(cinfo, JERR_NOT_COMPILED);
#endif
} else {
    /* No context, just make it tall enough for one row group */
    prep->pub.pre_process_data = pre_process_data;
    for (ci = 0, compptr = cinfo->comp_info; ci < cinfo->num_components;
        ci++, compptr++) {
        prep->color_buf[ci] = (*cinfo->mem->alloc_sarray)
            ((j_common_ptr) cinfo, JPOOL_IMAGE,
            (JDIMENSION) ((long) compptr->width_in_blocks * DCTSIZE *
                cinfo->max_h_samp_factor) / compptr->h_samp_factor),
            (JDIMENSION) cinfo->max_v_samp_factor);
    }
}

```

```

/*
 * jcsample.c
 *
 * Copyright (C) 1991-1996, Thomas G. Lane.
 * This file is part of the Independent JPEG Group's software.
 * For conditions of distribution and use, see the accompanying README file.
 *
 * This file contains downsampling routines.
 *
 * Downsampling input data is counted in "row groups".  A row group
 * is defined to be max_v_samp_factor pixel rows of each component,
 * from which the downsampler produces v_samp_factor sample rows.
 * A single row group is processed in each call to the downsampler module.
 *
 * The downsampler is responsible for edge-expansion of its output data
 * to fill an integral number of DCT blocks horizontally.  The source buffer
 * may be modified if it is helpful for this purpose (the source buffer is
 * allocated wide enough to correspond to the desired output width).
 * The caller (the prep controller) is responsible for vertical padding.
 *
 * The downsampler may request "context rows" by setting need_context_rows
 * during startup.  In this case, the input arrays will contain at least
 * one row group's worth of pixels above and below the passed-in data;
 * the caller will create dummy rows at image top and bottom by replicating
 * the first or last real pixel row.
 *
 * An excellent reference for image resampling is
 *   Digital Image Warping, George Wolberg, 1990.
 *   Pub. by IEEE Computer Society Press, Los Alamitos, CA. ISBN 0-8186-8944-7.
 *
 * The downsampling algorithm used here is a simple average of the source
 * pixels covered by the output pixel.  The hi-falutin sampling literature
 * refers to this as a "box filter".  In general the characteristics of a box
 * filter are not very good, but for the specific cases we normally use (1:1
 * and 2:1 ratios) the box is equivalent to a "triangle filter" which is not
 * nearly so bad.  If you intend to use other sampling ratios, you'd be well
 * advised to improve this code.
 *
 * A simple input-smoothing capability is provided.  This is mainly intended
 * for cleaning up color-dithered GIF input files (if you find it inadequate,
 * we suggest using an external filtering program such as pnmconvol).  When
 * enabled, each input pixel P is replaced by a weighted sum of itself and its
 * eight neighbors.  P's weight is 1-8*SF and each neighbor's weight is SF,
 * where SF = (smoothing_factor / 1024).
 * Currently, smoothing is only supported for 2h2v sampling factors.
 */

#define JPEG_INTERNALS
#include "jinclude.h"
#include "jpeglib.h"

/* Pointer to routine to downsample a single component */
typedef JMETHOD(void, downsample1_ptr,
  (j_compress_ptr cinfo, jpeg_component_info * comp_ptr,
   JSAMPARRAY input_data, JSAMPARRAY output_data));

/* Private subobject */

typedef struct {
  struct jpeg_downsampler pub; /* public fields */

  /* Downsampling method pointers, one per component */
  downsample1_ptr methods[MAX_COMPONENTS];
} my_downsampler;

typedef my_downsampler * my_downsample_ptr;

/*
 * Initialize for a downsampling pass.
 */

METHODDEF(void)
start_pass_downsample (j_compress_ptr cinfo)
{
  /* no work for now */
}

```

```

/*
 * Expand a component horizontally from width input_cols to width output_cols,
 * by duplicating the rightmost samples.
 */

LOCAL(void)
expand_right_edge (JSAMPARRAY image_data, int num_rows,
                  JDIMENSION input_cols, JDIMENSION output_cols)
{
    register JSAMPROW ptr;
    register JSAMPLE pixval;
    register int count;
    int row;
    int numcols = (int) (output_cols - input_cols);

    if (numcols > 0) {
        for (row = 0; row < num_rows; row++) {
            ptr = image_data[row] + input_cols;
            pixval = ptr[-1]; /* don't need GETJSAMPLE() here */
            for (count = numcols; count > 0; count--)
                *ptr++ = pixval;
        }
    }
}

/*
 * Do downsampling for a whole row group (all components).
 *
 * In this version we simply downsample each component independently.
 */

METHODDEF(void)
separate_downsample (j_compress_ptr cinfo,
                    JSAMPIMAGE input_buf, JDIMENSION in_row_index,
                    JSAMPIMAGE output_buf, JDIMENSION out_row_group_index)
{
    my_downsample_ptr downsample = (my_downsample_ptr) cinfo->downsample;
    int ci;
    jpeg_component_info * comp_ptr;
    JSAMPARRAY in_ptr, out_ptr;

    for (ci = 0, comp_ptr = cinfo->comp_info; ci < cinfo->num_components;
         ci++, comp_ptr++) {
        in_ptr = input_buf[ci] + in_row_index;
        out_ptr = output_buf[ci] + (out_row_group_index * comp_ptr->v_samp_factor);
        (*downsample->methods[ci]) (cinfo, comp_ptr, in_ptr, out_ptr);
    }
}

/*
 * Downsample pixel values of a single component.
 * One row group is processed per call.
 * This version handles arbitrary integral sampling ratios, without smoothing.
 * Note that this version is not actually used for customary sampling ratios.
 */

METHODDEF(void)
int_downsample (j_compress_ptr cinfo, jpeg_component_info * comp_ptr,
               JSAMPARRAY input_data, JSAMPARRAY output_data)
{
    int inrow, outrow, h_expand, v_expand, numpix, numpix2, h, v;
    JDIMENSION outcol, outcol_h; /* outcol_h == outcol*h_expand */
    JDIMENSION output_cols = comp_ptr->width_in_blocks * DCTSIZE;
    JSAMPROW inptr, outptr;
    INT32 outvalue;

    h_expand = cinfo->max_h_samp_factor / comp_ptr->h_samp_factor;
    v_expand = cinfo->max_v_samp_factor / comp_ptr->v_samp_factor;
    numpix = h_expand * v_expand;
    numpix2 = numpix/2;

    /* Expand input data enough to let all the output samples be generated
     * by the standard loop. Special-casing padded output would be more
     * efficient.
     */
    expand_right_edge(input_data, cinfo->max_v_samp_factor,
                    cinfo->image_width, output_cols * h_expand);
}

```

```

inrow = 0;
for (outrow = 0; outrow < comptr->v_samp_factor; outrow++) {
    outptr = output_data[outrow];
    for (outcol = 0, outcol_h = 0; outcol < output_cols;
        outcol++, outcol_h += h_expand) {
        outvalue = 0;
        for (v = 0; v < v_expand; v++) {
            inptr = input_data[inrow+v] + outcol_h;
            for (h = 0; h < h_expand; h++) {
                outvalue += (INT32) GETJSAMPLE(*inptr++);
            }
            *outptr++ = (JSAMPLE) ((outvalue + numpix2) / numpix);
        }
        inrow += v_expand;
    }
}

/*
 * Downsample pixel values of a single component.
 * This version handles the special case of a full-size component,
 * without smoothing.
 */

METHODDEF(void)
fullsize_downsample (j_compress_ptr cinfo, jpeg_component_info * comptr,
    JSAMPARRAY input_data, JSAMPARRAY output_data)
{
    /* Copy the data */
    jcopy_sample_rows(input_data, 0, output_data, 0,
        cinfo->max_v_samp_factor, cinfo->image_width);
    /* Edge-expand */
    expand_right_edge(output_data, cinfo->max_v_samp_factor,
        cinfo->image_width, comptr->width_in_blocks * DCTSIZE);
}

/*
 * Downsample pixel values of a single component.
 * This version handles the common case of 2:1 horizontal and 1:1 vertical,
 * without smoothing.
 *
 * A note about the "bias" calculations: when rounding fractional values to
 * integer, we do not want to always round 0.5 up to the next integer.
 * If we did that, we'd introduce a noticeable bias towards larger values.
 * Instead, this code is arranged so that 0.5 will be rounded up or down at
 * alternate pixel locations (a simple ordered dither pattern).
 */

METHODDEF(void)
h2v1_downsample (j_compress_ptr cinfo, jpeg_component_info * comptr,
    JSAMPARRAY input_data, JSAMPARRAY output_data)
{
    int outrow;
    JDIMENSION outcol;
    JDIMENSION output_cols = comptr->width_in_blocks * DCTSIZE;
    register JSAMPROW inptr, outptr;
    register int bias;

    /* Expand input data enough to let all the output samples be generated
     * by the standard loop. Special-casing padded output would be more
     * efficient.
     */
    expand_right_edge(input_data, cinfo->max_v_samp_factor,
        cinfo->image_width, output_cols * 2);

    for (outrow = 0; outrow < comptr->v_samp_factor; outrow++) {
        outptr = output_data[outrow];
        inptr = input_data[outrow];
        bias = 0; /* bias = 0,1,0,1,... for successive samples */
        for (outcol = 0; outcol < output_cols; outcol++) {
            *outptr++ = (JSAMPLE) ((GETJSAMPLE(*inptr) + GETJSAMPLE(inptr[1])
                + bias) >> 1);
            bias ^= 1; /* 0=>1, 1=>0 */
            inptr += 2;
        }
    }
}

```

```

/*
 * Downsample pixel values of a single component.
 * This version handles the standard case of 2:1 horizontal and 2:1 vertical,
 * without smoothing.
 */

```

```

METHODDEF(void)
h2v2_downsample (j_compress_ptr cinfo, jpeg_component_info * comp_ptr,
                 JSAMPARRAY input_data, JSAMPARRAY output_data)
{
    int inrow, outrow;
    JDIMENSION outcol;
    JDIMENSION output_cols = comp_ptr->width_in_blocks * DCTSIZE;
    register JSAMPROW inptr0, inptr1, outptr;
    register int bias;

```

```

    /* Expand input data enough to let all the output samples be generated
     * by the standard loop. Special-casing padded output would be more
     * efficient.
     */

```

```

    expand_right_edge(input_data, cinfo->max_v_samp_factor,
                     cinfo->image_width, output_cols * 2);

```

```

    inrow = 0;
    for (outrow = 0; outrow < comp_ptr->v_samp_factor; outrow++) {
        outptr = output_data[outrow];
        inptr0 = input_data[inrow];
        inptr1 = input_data[inrow+1];
        bias = 1; /* bias = 1,2,1,2,... for successive samples */
        for (outcol = 0; outcol < output_cols; outcol++) {
            *outptr++ = (JSAMPLE) ((GETJSAMPLE(*inptr0) + GETJSAMPLE(inptr0[1]) +
                                   GETJSAMPLE(*inptr1) + GETJSAMPLE(inptr1[1])
                                   + bias) >> 2);
            bias ^= 3; /* 1=>2, 2=>1 */
            inptr0 += 2; inptr1 += 2;
        }
        inrow += 2;
    }

```

```

#ifdef INPUT_SMOOTHING_SUPPORTED

```

```

/*
 * Downsample pixel values of a single component.
 * This version handles the standard case of 2:1 horizontal and 2:1 vertical,
 * with smoothing. One row of context is required.
 */

```

```

METHODDEF(void)
h2v2_smooth_downsample (j_compress_ptr cinfo, jpeg_component_info * comp_ptr,
                       JSAMPARRAY input_data, JSAMPARRAY output_data)
{

```

```

    int inrow, outrow;
    JDIMENSION colctr;
    JDIMENSION output_cols = comp_ptr->width_in_blocks * DCTSIZE;
    register JSAMPROW inptr0, inptr1, above_ptr, below_ptr, outptr;
    INT32 membersum, neighsum, memberscale, neighscale;

```

```

    /* Expand input data enough to let all the output samples be generated
     * by the standard loop. Special-casing padded output would be more
     * efficient.
     */

```

```

    expand_right_edge(input_data - 1, cinfo->max_v_samp_factor + 2,
                     cinfo->image_width, output_cols * 2);

```

```

    /* We don't bother to form the individual "smoothed" input pixel values;
     * we can directly compute the output which is the average of the four
     * smoothed values. Each of the four member pixels contributes a fraction
     * (1-8*SF) to its own smoothed image and a fraction SF to each of the three
     * other smoothed pixels, therefore a total fraction (1-5*SF)/4 to the final
     * output. The four corner-adjacent neighbor pixels contribute a fraction
     * SF to just one smoothed pixel, or SF/4 to the final output; while the
     * eight edge-adjacent neighbors contribute SF to each of two smoothed
     * pixels, or SF/2 overall. In order to use integer arithmetic, these
     * factors are scaled by 2^16 = 65536.
     * Also recall that SF = smoothing_factor / 1024.
     */

```

```

memberscale = 16384 - cinfo->smoothing_factor * 80; /* scaled (1-SF)/4 */
neighscale = cinfo->smoothing_factor * 16; /* scaled SF/4 */

inrow = 0;
for (outrow = 0; outrow < compptr->v_samp_factor; outrow++) {
    outptr = output_data[outrow];
    inptr0 = input_data[inrow];
    inptr1 = input_data[inrow+1];
    above_ptr = input_data[inrow-1];
    below_ptr = input_data[inrow+2];

    /* Special case for first column: pretend column -1 is same as column 0 */
    membersum = GETJSAMPLE(*inptr0) + GETJSAMPLE(inptr0[1]) +
        GETJSAMPLE(*inptr1) + GETJSAMPLE(inptr1[1]);
    neighsum = GETJSAMPLE(*above_ptr) + GETJSAMPLE(above_ptr[1]) +
        GETJSAMPLE(*below_ptr) + GETJSAMPLE(below_ptr[1]) +
        GETJSAMPLE(*inptr0) + GETJSAMPLE(inptr0[2]) +
        GETJSAMPLE(*inptr1) + GETJSAMPLE(inptr1[2]);
    neighsum += neighsum;
    neighsum += GETJSAMPLE(*above_ptr) + GETJSAMPLE(above_ptr[2]) +
        GETJSAMPLE(*below_ptr) + GETJSAMPLE(below_ptr[2]);
    membersum = membersum * memberscale + neighsum * neighscale;
    *outptr++ = (JSAMPLE) ((membersum + 32768) >> 16);
    inptr0 += 2; inptr1 += 2; above_ptr += 2; below_ptr += 2;

    for (colctr = output_cols - 2; colctr > 0; colctr--) {
        /* sum of pixels directly mapped to this output element */
        membersum = GETJSAMPLE(*inptr0) + GETJSAMPLE(inptr0[1]) +
            GETJSAMPLE(*inptr1) + GETJSAMPLE(inptr1[1]);
        /* sum of edge-neighbor pixels */
        neighsum = GETJSAMPLE(*above_ptr) + GETJSAMPLE(above_ptr[1]) +
            GETJSAMPLE(*below_ptr) + GETJSAMPLE(below_ptr[1]) +
            GETJSAMPLE(inptr0[-1]) + GETJSAMPLE(inptr0[2]) +
            GETJSAMPLE(inptr1[-1]) + GETJSAMPLE(inptr1[2]);
        /* The edge-neighbors count twice as much as corner-neighbors */
        neighsum += neighsum;
        /* Add in the corner-neighbors */
        neighsum += GETJSAMPLE(above_ptr[-1]) + GETJSAMPLE(above_ptr[2]) +
            GETJSAMPLE(below_ptr[-1]) + GETJSAMPLE(below_ptr[2]);
        /* form final output scaled up by 2^16 */
        membersum = membersum * memberscale + neighsum * neighscale;
        /* round, descale and output it */
        *outptr++ = (JSAMPLE) ((membersum + 32768) >> 16);
        inptr0 += 2; inptr1 += 2; above_ptr += 2; below_ptr += 2;
    }

    /* Special case for last column */
    membersum = GETJSAMPLE(*inptr0) + GETJSAMPLE(inptr0[1]) +
        GETJSAMPLE(*inptr1) + GETJSAMPLE(inptr1[1]);
    neighsum = GETJSAMPLE(*above_ptr) + GETJSAMPLE(above_ptr[1]) +
        GETJSAMPLE(*below_ptr) + GETJSAMPLE(below_ptr[1]) +
        GETJSAMPLE(inptr0[-1]) + GETJSAMPLE(inptr0[1]) +
        GETJSAMPLE(inptr1[-1]) + GETJSAMPLE(inptr1[1]);
    neighsum += neighsum;
    neighsum += GETJSAMPLE(above_ptr[-1]) + GETJSAMPLE(above_ptr[1]) +
        GETJSAMPLE(below_ptr[-1]) + GETJSAMPLE(below_ptr[1]);
    membersum = membersum * memberscale + neighsum * neighscale;
    *outptr = (JSAMPLE) ((membersum + 32768) >> 16);

    inrow += 2;
}
}

/*
 * Downsample pixel values of a single component.
 * This version handles the special case of a full-size component,
 * with smoothing. One row of context is required.
 */

```

```

METHODDEF(void)
fullsize_smooth_downsample (j_compress_ptr cinfo, jpeg_component_info *compptr,
    JSAMPARRAY input_data, JSAMPARRAY output_data)
{
    int outrow;
    JDIMENSION colctr;
    JDIMENSION output_cols = compptr->width_in_blocks * DCTSIZE;
    register JSAMPROW inptr, above_ptr, below_ptr, outptr;
    INT32 membersum, neighsum, memberscale, neighscale;
    int colsum, lastcolsum, nextcolsum;
}

```

```

/* Expand input data enough to let all the output samples be generated
 * by the standard loop. Special-casing padded output would be more
 * efficient.
 */
expand_right_edge(input_data - 1, cinfo->max_v_samp_factor + 2,
                  cinfo->image_width, output_cols);

/* Each of the eight neighbor pixels contributes a fraction SF to the
 * smoothed pixel, while the main pixel contributes (1-8*SF). In order
 * to use integer arithmetic, these factors are multiplied by 2^16 = 65536.
 * Also recall that SF = smoothing_factor / 1024.
 */

memberscale = 65536L - cinfo->smoothing_factor * 512L; /* scaled 1-8*SF */
neighscale = cinfo->smoothing_factor * 64; /* scaled SF */

for (outrow = 0; outrow < comp_ptr->v_samp_factor; outrow++) {
    outptr = output_data[outrow];
    inptr = input_data[outrow];
    above_ptr = input_data[outrow-1];
    below_ptr = input_data[outrow+1];

    /* Special case for first column */
    colsum = GETJSAMPLE(*above_ptr++) + GETJSAMPLE(*below_ptr++) +
             GETJSAMPLE(*inptr);
    membersum = GETJSAMPLE(*inptr++);
    nextcolsum = GETJSAMPLE(*above_ptr) + GETJSAMPLE(*below_ptr) +
                GETJSAMPLE(*inptr);
    neighsum = colsum + (colsum - membersum) + nextcolsum;
    membersum = membersum * memberscale + neighsum * neighscale;
    *outptr++ = (JSAMPLE) ((membersum + 32768) >> 16);
    lastcolsum = colsum; colsum = nextcolsum;

    for (colctr = output_cols - 2; colctr > 0; colctr--) {
        membersum = GETJSAMPLE(*inptr++);
        above_ptr++; below_ptr++;
        nextcolsum = GETJSAMPLE(*above_ptr) + GETJSAMPLE(*below_ptr) +
                    GETJSAMPLE(*inptr);
        neighsum = lastcolsum + (colsum - membersum) + nextcolsum;
        membersum = membersum * memberscale + neighsum * neighscale;
        *outptr++ = (JSAMPLE) ((membersum + 32768) >> 16);
        lastcolsum = colsum; colsum = nextcolsum;
    }

    /* Special case for last column */
    membersum = GETJSAMPLE(*inptr);
    neighsum = lastcolsum + (colsum - membersum) + colsum;
    membersum = membersum * memberscale + neighsum * neighscale;
    *outptr = (JSAMPLE) ((membersum + 32768) >> 16);
}

#endif /* INPUT_SMOOTHING_SUPPORTED */

/*
 * Module initialization routine for downsampling.
 * Note that we must select a routine for each component.
 */

GLOBAL(void)
jinit_downsampler (j_compress_ptr cinfo)
{
    my_downsample_ptr downsample;
    int ci;
    jpeg_component_info * comp_ptr;
    boolean smoothok = TRUE;

    downsample = (my_downsample_ptr)
        (*cinfo->mem->alloc_small) ((j_common_ptr) cinfo, JPOOL_IMAGE,
                                   sizeof(my_downsampler));
    cinfo->downsample = (struct jpeg_downsampler *) downsample;
    downsample->pub.start_pass = start_pass_downsample;
    downsample->pub.downsample = sep_downsample;
    downsample->pub.need_context_rows = FALSE;

    if (cinfo->CCIR601_sampling)
        ERREXIT(cinfo, JERR_CCIR601_NOTIMPL);
}

```

```

/* Verify we can handle the sampling factors, and set up method pointers */
for (ci = 0, compptr = cinfo->comp_info; ci < cinfo->num_components;
    ci++, compptr++) {
    if (compptr->h_samp_factor == cinfo->max_h_samp_factor &&
        compptr->v_samp_factor == cinfo->max_v_samp_factor) {
#ifdef INPUT_SMOOTHING_SUPPORTED
        if (cinfo->smoothing_factor) {
            downsample->methods[ci] = fullsize_smooth_downsample;
            downsample->pub.need_context_rows = TRUE;
        } else
#endif
        downsample->methods[ci] = fullsize_downsample;
    } else if (compptr->h_samp_factor * 2 == cinfo->max_h_samp_factor &&
        compptr->v_samp_factor == cinfo->max_v_samp_factor) {
        smoothok = FALSE;
        downsample->methods[ci] = h2v1_downsample;
    } else if (compptr->h_samp_factor * 2 == cinfo->max_h_samp_factor &&
        compptr->v_samp_factor * 2 == cinfo->max_v_samp_factor) {
#ifdef INPUT_SMOOTHING_SUPPORTED
        if (cinfo->smoothing_factor) {
            downsample->methods[ci] = h2v2_smooth_downsample;
            downsample->pub.need_context_rows = TRUE;
        } else
#endif
        downsample->methods[ci] = h2v2_downsample;
    } else if ((cinfo->max_h_samp_factor % compptr->h_samp_factor) == 0 &&
        (cinfo->max_v_samp_factor % compptr->v_samp_factor) == 0) {
        smoothok = FALSE;
        downsample->methods[ci] = int_downsample;
    } else
        ERREXIT(cinfo, JERR_FRACT_SAMPLE_NOTIMPL);
}

#ifdef INPUT_SMOOTHING_SUPPORTED
if (cinfo->smoothing_factor && !smoothok)
    TRACEEMS(cinfo, 0, JTRC_SMOOTH_NOTIMPL);
#endif
}

```



```

/*
 * jctrans.c
 *
 * Copyright (C) 1995-1998, Thomas G. Lane.
 * This file is part of the Independent JPEG Group's software.
 * For conditions of distribution and use, see the accompanying README file.
 *
 * This file contains library routines for transcoding compression,
 * that is, writing raw DCT coefficient arrays to an output JPEG file.
 * The routines in jcapimin.c will also be needed by a transcoder.
 */

#define JPEG_INTERNALS
#include "jinclude.h"
#include "jpeglib.h"

/* Forward declarations */
LOCAL(void) transencode_master_selection
    JPP((j_compress_ptr cinfo, jvirt_barray_ptr * coef_arrays));
LOCAL(void) transencode_coef_controller
    JPP((j_compress_ptr cinfo, jvirt_barray_ptr * coef_arrays));

/*
 * Compression initialization for writing raw-coefficient data.
 * Before calling this, all parameters and a data destination must be set up.
 * Call jpeg_finish_compress() to actually write the data.
 *
 * The number of passed virtual arrays must match cinfo->num_components.
 * Note that the virtual arrays need not be filled or even realized at
 * the time write_coefficients is called; indeed, if the virtual arrays
 * were requested from this compression object's memory manager, they
 * typically will be realized during this routine and filled afterwards.
 */
GLOBAL(void)
jpeg_write_coefficients (j_compress_ptr cinfo, jvirt_barray_ptr * coef_arrays)
{
    if (cinfo->global_state != CSTATE_START)
        ERREXIT1(cinfo, JERR_BAD_STATE, cinfo->global_state);
    /* Mark all tables to be written */
    jpeg_suppress_tables(cinfo, FALSE);
    /* (Re)initialize error mgr and destination modules */
    (*cinfo->err->reset_error_mgr) ((j_common_ptr) cinfo);
    (*cinfo->dest->init_destination) (cinfo);
    /* Perform master selection of active modules */
    transencode_master_selection(cinfo, coef_arrays);
    /* Wait for jpeg_finish_compress() call */
    cinfo->next_scanline = 0; /* so jpeg_write_marker works */
    cinfo->global_state = CSTATE_WRCOEFS;
}

/*
 * Initialize the compression object with default parameters,
 * then copy from the source object all parameters needed for lossless
 * transcoding. Parameters that can be varied without loss (such as
 * scan script and Huffman optimization) are left in their default states.
 */
GLOBAL(void)
jpeg_copy_critical_parameters (j_decompress_ptr srcinfo,
                              j_compress_ptr dstinfo)
{
    JQUANT_TBL ** qtblptr;
    jpeg_component_info *incomp, *outcomp;
    JQUANT_TBL *c_quant, *slot_quant;
    int tblno, ci, coefi;

    /* Safety check to ensure start_compress not called yet. */
    if (dstinfo->global_state != CSTATE_START)
        ERREXIT1(dstinfo, JERR_BAD_STATE, dstinfo->global_state);
    /* Copy fundamental image dimensions */
    dstinfo->image_width = srcinfo->image_width;
    dstinfo->image_height = srcinfo->image_height;
    dstinfo->input_components = srcinfo->num_components;
    dstinfo->in_color_space = srcinfo->jpeg_color_space;
    /* Initialize all parameters to default values */
    jpeg_set_defaults(dstinfo);
}

```

```

/* jpeg_set_defaults may choose wrong colorspace, eg YCbCr if input is RGB.
 * Fix it to get the right color markers for the image colorspace.
 */
jpeg_set_colorspace(dstinfo, srcinfo->jpeg_color_space);
dstinfo->data_precision = srcinfo->data_precision;
dstinfo->CCIR601_sampling = srcinfo->CCIR601_sampling;
/* Copy the source's quantization tables. */
for (tblno = 0; tblno < NUM_QUANT_TBLS; tblno++) {
    if (srcinfo->quant_tbl_ptrs[tblno] != NULL) {
        qtblptr = &dstinfo->quant_tbl_ptrs[tblno];
        if (*qtblptr == NULL)
            *qtblptr = jpeg_alloc_quant_table((j_common_ptr) dstinfo);
        MEMCOPY((*qtblptr)->quantval,
            srcinfo->quant_tbl_ptrs[tblno]->quantval,
            SIZEOF((*qtblptr)->quantval));
        (*qtblptr)->sent_table = FALSE;
    }
}
/* Copy the source's per-component info.
 * Note we assume jpeg_set_defaults has allocated the dest comp_info array.
 */
dstinfo->num_components = srcinfo->num_components;
if (dstinfo->num_components < 1 || dstinfo->num_components > MAX_COMPONENTS)
    ERREXIT2(dstinfo, JERR_COMPONENT_COUNT, dstinfo->num_components,
        MAX_COMPONENTS);
for (ci = 0, incomp = srcinfo->comp_info, outcomp = dstinfo->comp_info;
    ci < dstinfo->num_components; ci++, incomp++, outcomp++) {
    outcomp->component_id = incomp->component_id;
    outcomp->h_samp_factor = incomp->h_samp_factor;
    outcomp->v_samp_factor = incomp->v_samp_factor;
    outcomp->quant_tbl_no = incomp->quant_tbl_no;
    /* Make sure saved quantization table for component matches the qtable
     * slot. If not, the input file re-used this qtable slot.
     * IJG encoder currently cannot duplicate this.
     */
    tblno = outcomp->quant_tbl_no;
    if (tblno < 0 || tblno >= NUM_QUANT_TBLS ||
        srcinfo->quant_tbl_ptrs[tblno] == NULL)
        ERREXIT1(dstinfo, JERR_NO_QUANT_TABLE, tblno);
    slot_quant = srcinfo->quant_tbl_ptrs[tblno];
    c_quant = incomp->quant_table;
    if (c_quant != NULL) {
        for (coefi = 0; coefi < DCTSIZE2; coefi++) {
            if (c_quant->quantval[coefi] != slot_quant->quantval[coefi])
                ERREXIT1(dstinfo, JERR_MISMATCHED_QUANT_TABLE, tblno);
        }
    }
    /* Note: we do not copy the source's Huffman table assignments;
     * instead we rely on jpeg_set_colorspace to have made a suitable choice.
     */
    /* Also copy JFIF version and resolution information, if available.
     * Strictly speaking this isn't "critical" info, but it's nearly
     * always appropriate to copy it if available. In particular,
     * if the application chooses to copy JFIF 1.02 extension markers from
     * the source file, we need to copy the version to make sure we don't
     * emit a file that has 1.02 extensions but a claimed version of 1.01.
     * We will *not*, however, copy version info from mislabeled "2.01" files.
     */
    if (srcinfo->saw_JFIF_marker) {
        if (srcinfo->JFIF_major_version == 1) {
            dstinfo->JFIF_major_version = srcinfo->JFIF_major_version;
            dstinfo->JFIF_minor_version = srcinfo->JFIF_minor_version;
        }
        dstinfo->density_unit = srcinfo->density_unit;
        dstinfo->X_density = srcinfo->X_density;
        dstinfo->Y_density = srcinfo->Y_density;
    }
}

/*
 * Master selection of compression modules for transcoding.
 * This substitutes for jcinit.c's initialization of the full compressor.
 */
LOCAL(void)
transcode_master_selection (j_compress_ptr cinfo,
    jvirt_barray_ptr * coef_arrays)
{

```

```

/* Although we don't actually use input_components for transcoding,
 * jcmaster.c's initial_setup will complain if input_components
 */
cinfo->input_components = 1;
/* Initialize master control (includes parameter checking/processing) */
jinit_c_master_control(cinfo, TRUE /* transcode only */);

/* Entropy encoding: either Huffman or arithmetic coding. */
if (cinfo->arith_code) {
    ERREXIT(cinfo, JERR_ARITH_NOTIMPL);
} else {
    if (cinfo->progressive_mode) {
#ifdef C_PROGRESSIVE_SUPPORTED
        jinit_phuff_encoder(cinfo);
    #else
        ERREXIT(cinfo, JERR_NOT_COMPILED);
    #endif
    } else
        jinit_huff_encoder(cinfo);
}

/* We need a special coefficient buffer controller. */
transcode_coef_controller(cinfo, coef_arrays);

jinit_marker_writer(cinfo);

/* We can now tell the memory manager to allocate virtual arrays. */
(*cinfo->mem->realize_virt_arrays) ((j_common_ptr) cinfo);

/* Write the datastream header (SOI, JFIF) immediately.
 * Frame and scan headers are postponed till later.
 * This lets application insert special markers after the SOI.
 */
(*cinfo->marker->write_file_header) (cinfo);
}

/*
 * The rest of this file is a special implementation of the coefficient
 * buffer controller. This is similar to jccoefct.c, but it handles only
 * output from presupplied virtual arrays. Furthermore, we generate any
 * dummy padding blocks on-the-fly rather than expecting them to be present
 * in the arrays.
 */
/* Private buffer controller object */
typedef struct {
    struct jpeg_c_coef_controller pub; /* public fields */
    JDIMENSION iMCU_row_num; /* iMCU row # within image */
    JDIMENSION mcu_ctr; /* counts MCUs processed in current row */
    int MCU_vert_offset; /* counts MCU rows within iMCU row */
    int MCU_rows_per_iMCU_row; /* number of such rows needed */

    /* Virtual block array for each component. */
    jvirt_barray_ptr * whole_image;

    /* Workspace for constructing dummy blocks at right/bottom edges. */
    JBLOCKROW dummy_buffer[C_MAX_BLOCKS_IN_MCU];
} my_coef_controller;

typedef my_coef_controller * my_coef_ptr;

LOCAL(void)
start_iMCU_row (j_compress_ptr cinfo)
/* Reset within-iMCU-row counters for a new row */
{
    my_coef_ptr coef = (my_coef_ptr) cinfo->coef;

    /* In an interleaved scan, an MCU row is the same as an iMCU row.
     * In a noninterleaved scan, an iMCU row has v_samp_factor MCU rows.
     * But at the bottom of the image, process only what's left.
     */
    if (cinfo->comps_in_scan > 1) {
        coef->MCU_rows_per_iMCU_row = 1;
    } else {
        if (coef->iMCU_row_num < (cinfo->total_iMCU_rows-1))
            coef->MCU_rows_per_iMCU_row = cinfo->cur_comp_info[0]->v_samp_factor;
    }
}

```

```

    else
        coef->MCU_rows_per_iMCU = cinfo->cur_comp_info[0]->last_row_height;
    }

    coef->mcu_ctr = 0;
    coef->MCU_vert_offset = 0;
}

/*
 * Initialize for a processing pass.
 */

METHODDEF(void)
start_pass_coef (j_compress_ptr cinfo, J_BUF_MODE pass_mode)
{
    my_coef_ptr coef = (my_coef_ptr) cinfo->coef;

    if (pass_mode != JBUF_CRANK_DEST)
        ERREXIT(cinfo, JERR_BAD_BUFFER_MODE);

    coef->iMCU_row_num = 0;
    start_iMCU_row(cinfo);
}

/*
 * Process some data.
 * We process the equivalent of one fully interleaved MCU row ("iMCU" row)
 * per call, ie, v_samp_factor block rows for each component in the scan.
 * The data is obtained from the virtual arrays and fed to the entropy coder.
 * Returns TRUE if the iMCU row is completed, FALSE if suspended.
 * NB: input_buf is ignored; it is likely to be a NULL pointer.
 */

METHODDEF(boolean)
compress_output (j_compress_ptr cinfo, JSAMPIMAGE input_buf)
{
    my_coef_ptr coef = (my_coef_ptr) cinfo->coef;
    JDIMENSION MCU_col_num; /* index of current MCU within row */
    JDIMENSION last_MCU_col = cinfo->MCUs_per_row - 1;
    JDIMENSION last_iMCU_row = cinfo->total_iMCU_rows - 1;
    int blkcn, ci, xindex, yindex, yoffset, blockcnt;
    JDIMENSION start_col;
    JBLOCKARRAY buffer[MAX_COMPS_IN_SCAN];
    JBLOCKROW MCU_buffer[C_MAX_BLOCKS_IN_MCU];
    JBLOCKROW buffer_ptr;
    jpeg_component_info *comp_ptr;

    /* Align the virtual buffers for the components used in this scan. */
    for (ci = 0; ci < cinfo->comps_in_scan; ci++) {
        comp_ptr = cinfo->cur_comp_info[ci];
        buffer[ci] = (*cinfo->mem->access_virt_barray)
            ((j_common_ptr) cinfo, coef->whole_image[comp_ptr->component_index],
             coef->iMCU_row_num * comp_ptr->v_samp_factor,
             (JDIMENSION) comp_ptr->v_samp_factor, FALSE);
    }

    /* Loop to process one whole iMCU row */
    for (yoffset = coef->MCU_vert_offset; yoffset < coef->MCU_rows_per_iMCU_row;
         yoffset++) {
        for (MCU_col_num = coef->mcu_ctr; MCU_col_num < cinfo->MCUs_per_row;
             MCU_col_num++) {
            /* Construct list of pointers to DCT blocks belonging to this MCU */
            blkcn = 0; /* index of current DCT block within MCU */
            for (ci = 0; ci < cinfo->comps_in_scan; ci++) {
                comp_ptr = cinfo->cur_comp_info[ci];
                start_col = MCU_col_num * comp_ptr->MCU_width;
                blockcnt = (MCU_col_num < last_MCU_col) ? comp_ptr->MCU_width
                    : comp_ptr->last_col_width;
                for (yindex = 0; yindex < comp_ptr->MCU_height; yindex++) {
                    if (coef->iMCU_row_num < last_iMCU_row ||
                        yindex+yoffset < comp_ptr->last_row_height) {
                        /* Fill in pointers to real blocks in this row */
                        buffer_ptr = buffer[ci][yindex+yoffset] + start_col;
                        for (xindex = 0; xindex < blockcnt; xindex++)
                            MCU_buffer[blkcn++] = buffer_ptr++;
                    } else {
                        /* At bottom of image, need a whole row of dummy blocks */

```

```

    xindex = 0;
}
/* Fill in any dummy blocks needed in this row.
 * Dummy blocks are filled in the same way as in jccoefct.c:
 * all zeroes in the AC entries, DC entries equal to previous
 * block's DC value. The init routine has already zeroed the
 * AC entries, so we need only set the DC entries correctly.
 */
for (; xindex < compptr->MCU_width; xindex++) {
    MCU_buffer[blkn] = coef->dummy_buffer[blkn];
    MCU_buffer[blkn][0][0] = MCU_buffer[blkn-1][0][0];
    blkn++;
}
}
/* Try to write the MCU. */
if (! (*cinfo->entropy->encode_mcu) (cinfo, MCU_buffer)) {
/* Suspension forced; update state counters and exit */
coef->MCU_vert_offset = yoffset;
coef->mcu_ctr = MCU_col_num;
return FALSE;
}
/* Completed an MCU row, but perhaps not an iMCU row */
coef->mcu_ctr = 0;
}
/* Completed the iMCU row, advance counters for next one */
coef->iMCU_row_num++;
start_iMCU_row(cinfo);
return TRUE;
}

/*
 * Initialize coefficient buffer controller.
 *
 * Each passed coefficient array must be the right size for that
 * coefficient: width_in_blocks wide and height_in_blocks high,
 * with unitheight at least v_samp_factor.
 */
LOCAL(void)
transcode_coef_controller (j_compress_ptr cinfo,
                           jvirt_barray_ptr * coef_arrays)
{
    my_coef_ptr coef;
    JBLOCKROW buffer;
    int i;

    coef = (my_coef_ptr)
        (*cinfo->mem->alloc_small) ((j_common_ptr) cinfo, JPOOL_IMAGE,
                                   sizeof(my_coef_controller));
    cinfo->coef = (struct jpeg_c_coef_controller *) coef;
    coef->pub.start_pass = start_pass_coef;
    coef->pub.compress_data = compress_output;

    /* Save pointer to virtual arrays */
    coef->whole_image = coef_arrays;

    /* Allocate and pre-zero space for dummy DCT blocks. */
    buffer = (JBLOCKROW)
        (*cinfo->mem->alloc_large) ((j_common_ptr) cinfo, JPOOL_IMAGE,
                                   C_MAX_BLOCKS_IN_MCU * sizeof(JBLOCK));
    jzero_far((void *) buffer, C_MAX_BLOCKS_IN_MCU * sizeof(JBLOCK));
    for (i = 0; i < C_MAX_BLOCKS_IN_MCU; i++) {
        coef->dummy_buffer[i] = buffer + i;
    }
}

```

```

/*
 * jdapimin.c
 *
 * Copyright (C) 1994-1998, Thomas G. Lane.
 * This file is part of the Independent JPEG Group's software.
 * For conditions of distribution and use, see the accompanying README file.
 *
 * This file contains application interface code for the decompression half
 * of the JPEG library. These are the "minimum" API routines that may be
 * needed in either the normal full-decompression case or the
 * transcoding-only case.
 *
 * Most of the routines intended to be called directly by an application
 * are in this file or in jdapistd.c. But also see jcomapi.c for routines
 * shared by compression and decompression, and jdtrans.c for the transcoding
 * case.
 */

#define JPEG_INTERNALS
#include "jinclude.h"
#include "jpeglib.h"

/*
 * Initialization of a JPEG decompression object.
 * The error manager must already be set up (in case memory manager fails).
 */

GLOBAL(void)
jpeg_CreateDecompress (j_decompress_ptr cinfo, int version, size_t structsize)
{
  int i;

  /* Guard against version mismatches between library and caller. */
  cinfo->mem = NULL; /* so jpeg_destroy knows mem mgr not called */
  if (version != JPEG_LIB_VERSION)
    ERREXIT2(cinfo, JERR_BAD_LIB_VERSION, JPEG_LIB_VERSION, version);
  if (structsize != SIZEOF(struct jpeg_decompress_struct))
    ERREXIT2(cinfo, JERR_BAD_STRUCT_SIZE,
              (int) SIZEOF(struct jpeg_decompress_struct), (int) structsize);

  /* For debugging purposes, we zero the whole master structure.
   * But the application has already set the err pointer, and may have set
   * client_data, so we have to save and restore those fields.
   * Note: if application hasn't set client_data, tools like Purify may
   * complain here.
   */
  struct jpeg_error_mgr * err = cinfo->err;
  void * client_data = cinfo->client_data; /* ignore Purify complaint here */
  MEMZERO(cinfo, SIZEOF(struct jpeg_decompress_struct));
  cinfo->err = err;
  cinfo->client_data = client_data;
}

cinfo->is_decompressor = TRUE;

/* Initialize a memory manager instance for this object */
jinit_memory_mgr((j_common_ptr) cinfo);

/* Zero out pointers to permanent structures. */
cinfo->progress = NULL;
cinfo->src = NULL;

for (i = 0; i < NUM_QUANT_TBLS; i++)
  cinfo->quant_tbl_ptrs[i] = NULL;

for (i = 0; i < NUM_HUFF_TBLS; i++) {
  cinfo->dc_huff_tbl_ptrs[i] = NULL;
  cinfo->ac_huff_tbl_ptrs[i] = NULL;
}

/* Initialize marker processor so application can override methods
 * for COM, APPn markers before calling jpeg_read_header.
 */
cinfo->marker_list = NULL;
jinit_marker_reader(cinfo);

/* And initialize the overall input controller. */
jinit_input_controller(cinfo);

```

```

/* OK, I'm ready */
cinfo->global_state = DSTART;
}

/*
 * Destruction of a JPEG decompression object
 */

GLOBAL(void)
jpeg_destroy_decompress (j_decompress_ptr cinfo)
{
    jpeg_destroy((j_common_ptr) cinfo); /* use common routine */
}

/*
 * Abort processing of a JPEG decompression operation,
 * but don't destroy the object itself.
 */

GLOBAL(void)
jpeg_abort_decompress (j_decompress_ptr cinfo)
{
    jpeg_abort((j_common_ptr) cinfo); /* use common routine */
}

/*
 * Set default decompression parameters.
 */

LOCAL(void)
default_decompress_parms (j_decompress_ptr cinfo)
{
    /* Guess the input colorspace, and set output colorspace accordingly. */
    /* (Wish JPEG committee had provided a real way to specify this...) */
    /* Note application may override our guesses. */
    switch (cinfo->num_components) {
        case 1:
            cinfo->jpeg_color_space = JCS_GRAYSCALE;
            cinfo->out_color_space = JCS_GRAYSCALE;
            break;

        case 3:
            if (cinfo->saw_JFIF_marker) {
                cinfo->jpeg_color_space = JCS_YCbCr; /* JFIF implies YCbCr */
            } else if (cinfo->saw_Adobe_marker) {
                switch (cinfo->Adobe_transform) {
                    case 0:
                        cinfo->jpeg_color_space = JCS_RGB;
                        break;
                    case 1:
                        cinfo->jpeg_color_space = JCS_YCbCr;
                        break;
                    default:
                        WARNMS1(cinfo, JWRN_ADOBE_XFORM, cinfo->Adobe_transform);
                        cinfo->jpeg_color_space = JCS_YCbCr; /* assume it's YCbCr */
                        break;
                }
            } else {
                /* Saw no special markers, try to guess from the component IDs */
                int cid0 = cinfo->comp_info[0].component_id;
                int cid1 = cinfo->comp_info[1].component_id;
                int cid2 = cinfo->comp_info[2].component_id;

                if (cid0 == 1 && cid1 == 2 && cid2 == 3)
                    cinfo->jpeg_color_space = JCS_YCbCr; /* assume JFIF w/out marker */
                else if (cid0 == 82 && cid1 == 71 && cid2 == 66)
                    cinfo->jpeg_color_space = JCS_RGB; /* ASCII 'R', 'G', 'B' */
                else {
                    TRACE3(cinfo, 1, JTRC_UNKNOWN_IDS, cid0, cid1, cid2);
                    cinfo->jpeg_color_space = JCS_YCbCr; /* assume it's YCbCr */
                }
            }
            /* Always guess RGB is proper output colorspace. */
            cinfo->out_color_space = JCS_RGB;
            break;

        case 4:

```

```

if (cinfo->saw_Adobe_marker) {
    switch (cinfo->Adobe_transform) {
        case 0:
            cinfo->jpeg_color_space = JCS_CMYK;
            break;
        case 2:
            cinfo->jpeg_color_space = JCS_YCCK;
            break;
        default:
            WARNMS1(cinfo, JWRN_ADOBE_XFORM, cinfo->Adobe_transform);
            cinfo->jpeg_color_space = JCS_YCCK; /* assume it's YCCK */
            break;
    }
} else {
    /* No special markers, assume straight CMYK. */
    cinfo->jpeg_color_space = JCS_CMYK;
}
cinfo->out_color_space = JCS_CMYK;
break;

default:
    cinfo->jpeg_color_space = JCS_UNKNOWN;
    cinfo->out_color_space = JCS_UNKNOWN;
    break;
}

/* Set defaults for other decompression parameters. */
cinfo->scale_num = 1; /* 1:1 scaling */
cinfo->scale_denom = 1;
cinfo->output_gamma = 1.0;
cinfo->buffered_image = FALSE;
cinfo->raw_data_out = FALSE;
cinfo->dct_method = JDCT_DEFAULT;
cinfo->do_fancy_upsampling = TRUE;
cinfo->do_block_smoothing = TRUE;
cinfo->quantize_colors = FALSE;
/* We set these in case application only sets quantize_colors. */
cinfo->dither_mode = JDITHER_FS;
#ifdef QUANT_2PASS_SUPPORTED
cinfo->two_pass_quantize = TRUE;
#else
cinfo->two_pass_quantize = FALSE;
#endif
cinfo->desired_number_of_colors = 256;
cinfo->colormap = NULL;
/* Initialize for no mode change in buffered-image mode. */
cinfo->enable_1pass_quant = FALSE;
cinfo->enable_external_quant = FALSE;
cinfo->enable_2pass_quant = FALSE;
}

/*
 * Decompression startup: read start of JPEG datastream to see what's there.
 * Need only initialize JPEG object and supply a data source before calling.
 *
 * This routine will read as far as the first SOS marker (ie, actual start of
 * compressed data), and will save all tables and parameters in the JPEG
 * object. It will also initialize the decompression parameters to default
 * values, and finally return JPEG_HEADER_OK. On return, the application may
 * adjust the decompression parameters and then call jpeg_start_decompress.
 * (Or, if the application only wanted to determine the image parameters,
 * the data need not be decompressed. In that case, call jpeg_abort or
 * jpeg_destroy to release any temporary space.)
 * If an abbreviated (tables only) datastream is presented, the routine will
 * return JPEG_HEADER_TABLES_ONLY upon reaching EOI. The application may then
 * re-use the JPEG object to read the abbreviated image datastream(s).
 * It is unnecessary (but OK) to call jpeg_abort in this case.
 * The JPEG_SUSPENDED return code only occurs if the data source module
 * requests suspension of the decompressor. In this case the application
 * should load more source data and then re-call jpeg_read_header to resume
 * processing.
 * If a non-suspending data source is used and require_image is TRUE, then the
 * return code need not be inspected since only JPEG_HEADER_OK is possible.
 *
 * This routine is now just a front end to jpeg_consume_input, with some
 * extra error checking.
 */

```

GLOBAL(int)



```

        break;
    default:
        ERREXIT1(cinfo, JERR_BAD_STATE, cinfo->global_state);
    }
    return retcode;
}

/*
 * Have we finished reading the input file?
 */

GLOBAL(boolean)
jpeg_input_complete (j_decompress_ptr cinfo)
{
    /* Check for valid jpeg object */
    if (cinfo->global_state < DSTATE_START ||
        cinfo->global_state > DSTATE_STOPPING)
        ERREXIT1(cinfo, JERR_BAD_STATE, cinfo->global_state);
    return cinfo->inputctl->eoi_reached;
}

/*
 * Is there more than one scan?
 */

GLOBAL(boolean)
jpeg_has_multiple_scans (j_decompress_ptr cinfo)
{
    /* Only valid after jpeg_read_header completes */
    if (cinfo->global_state < DSTATE_READY ||
        cinfo->global_state > DSTATE_STOPPING)
        ERREXIT1(cinfo, JERR_BAD_STATE, cinfo->global_state);
    return cinfo->inputctl->has_multiple_scans;
}

/*
 * Finish JPEG decompression.
 *
 * This will normally just verify the file trailer and release temp storage.
 *
 * Returns FALSE if suspended. The return value need be inspected only if
 * a suspending data source is used.
 */

GLOBAL(boolean)
jpeg_finish_decompress (j_decompress_ptr cinfo)
{
    if ((cinfo->global_state == DSTATE_SCANNING ||
        cinfo->global_state == DSTATE_RAW_OK) && ! cinfo->buffered_image) {
        /* Terminate final pass of non-buffered mode */
        if (cinfo->output_scanline < cinfo->output_height)
            ERREXIT(cinfo, JERR_TOO_LITTLE_DATA);
        (*cinfo->master->finish_output_pass) (cinfo);
        cinfo->global_state = DSTATE_STOPPING;
    } else if (cinfo->global_state == DSTATE_BUFIMAGE) {
        /* Finishing after a buffered-image operation */
        cinfo->global_state = DSTATE_STOPPING;
    } else if (cinfo->global_state != DSTATE_STOPPING) {
        /* STOPPING = repeat call after a suspension, anything else is error */
        ERREXIT1(cinfo, JERR_BAD_STATE, cinfo->global_state);
    }
    /* Read until EOI */
    while (! cinfo->inputctl->eoi_reached) {
        if ((*cinfo->inputctl->consume_input) (cinfo) == JPEG_SUSPENDED)
            return FALSE; /* Suspend, come back later */
    }
    /* Do final cleanup */
    (*cinfo->src->term_source) (cinfo);
    /* We can use jpeg_abort to release memory and reset global_state */
    jpeg_abort((j_common_ptr) cinfo);
    return TRUE;
}

```

```

jpeg_read_header (j_decompress_ptr cinfo, boolean require_image)
{
    int retcode;

    if (cinfo->global_state != DSTATE_START &&
        cinfo->global_state != DSTATE_INHEADER)
        ERREXIT1(cinfo, JERR_BAD_STATE, cinfo->global_state);

    retcode = jpeg_consume_input(cinfo);

    switch (retcode) {
    case JPEG_REACHED_SOS:
        retcode = JPEG_HEADER_OK;
        break;
    case JPEG_REACHED_EOI:
        if (require_image) /* Complain if application wanted an image */
            ERREXIT(cinfo, JERR_NO_IMAGE);
        /* Reset to start state; it would be safer to require the application to
         * call jpeg_abort, but we can't change it now for compatibility reasons.
         * A side effect is to free any temporary memory (there shouldn't be any).
         */
        jpeg_abort((j_common_ptr) cinfo); /* sets state = DSTATE_START */
        retcode = JPEG_HEADER_TABLES_ONLY;
        break;
    case JPEG_SUSPENDED:
        /* no work */
        break;
    }
}

```

```

    return retcode;
}

```

Consume data in advance of what the decompressor requires.  
 This can be called at any time once the decompressor object has  
 been created and a data source has been set up.

This routine is essentially a state machine that handles a couple  
 of critical state-transition actions, namely initial setup and  
 transition from header scanning to ready-for-start\_decompress.  
 All the actual input is done via the input controller's consume\_input  
 method.

```

GLOBAL(int)
jpeg_consume_input (j_decompress_ptr cinfo)
{
    int retcode = JPEG_SUSPENDED;

    /* NB: every possible DSTATE value should be listed in this switch */
    switch (cinfo->global_state) {
    case DSTATE_START:
        /* Start-of-datastream actions: reset appropriate modules */
        (*cinfo->inputctl->reset_input_controller) (cinfo);
        /* Initialize application's data source module */
        (*cinfo->src->init_source) (cinfo);
        cinfo->global_state = DSTATE_INHEADER;
        /* FALLTHROUGH */
    case DSTATE_INHEADER:
        retcode = (*cinfo->inputctl->consume_input) (cinfo);
        if (retcode == JPEG_REACHED_SOS) { /* Found SOS, prepare to decompress */
            /* Set up default parameters based on header data */
            default_decompress_parms(cinfo);
            /* Set global state: ready for start_decompress */
            cinfo->global_state = DSTATE_READY;
        }
        break;
    case DSTATE_READY:
        /* Can't advance past first SOS until start_decompress is called */
        retcode = JPEG_REACHED_SOS;
        break;
    case DSTATE_PRELOAD:
    case DSTATE_PRESCAN:
    case DSTATE_SCANNING:
    case DSTATE_RAW_OK:
    case DSTATE_BUFIMAGE:
    case DSTATE_BUFPOST:
    case DSTATE_STOPPING:
        retcode = (*cinfo->inputctl->consume_input) (cinfo);
    }
}

```

```

/* Perform any dummy output passes, and set up for the final pass */
return output_pass_setup(cinfo);
}

/*
 * Set up for an output pass, and perform any dummy pass(es) needed.
 * Common subroutine for jpeg_start_decompress and jpeg_start_output.
 * Entry: global_state = DSTATE_PRESCAN only if previously suspended.
 * Exit: If done, returns TRUE and sets global_state for proper output mode.
 *       If suspended, returns FALSE and sets global_state = DSTATE_PRESCAN.
 */

LOCAL(boolean)
output_pass_setup (j_decompress_ptr cinfo)
{
    if (cinfo->global_state != DSTATE_PRESCAN) {
        /* First call: do pass setup */
        (*cinfo->master->prepare_for_output_pass) (cinfo);
        cinfo->output_scanline = 0;
        cinfo->global_state = DSTATE_PRESCAN;
    }
    /* Loop over any required dummy passes */
    while (cinfo->master->is_dummy_pass) {
#ifdef QUANT_2PASS_SUPPORTED
        /* Crank through the dummy pass */
        while (cinfo->output_scanline < cinfo->output_height) {
            JDIMENSION last_scanline;
            /* Call progress monitor hook if present */
            if (cinfo->progress != NULL) {
                cinfo->progress->pass_counter = (long) cinfo->output_scanline;
                cinfo->progress->pass_limit = (long) cinfo->output_height;
                (*cinfo->progress->progress_monitor) ((j_common_ptr) cinfo);
            }
            /* Process some data */
            last_scanline = cinfo->output_scanline;
            (*cinfo->main->process_data) (cinfo, (JSAMPARRAY) NULL,
                                         &cinfo->output_scanline, (JDIMENSION) 0);
            if (cinfo->output_scanline == last_scanline)
                return FALSE; /* No progress made, must suspend */
        }
        /* Finish up dummy pass, and set up for another one */
        (*cinfo->master->finish_output_pass) (cinfo);
        (*cinfo->master->prepare_for_output_pass) (cinfo);
        cinfo->output_scanline = 0;
#else
        ERREXIT(cinfo, JERR_NOT_COMPILED);
#endif /* QUANT_2PASS_SUPPORTED */
    }
    /* Ready for application to drive output pass through
     * jpeg_read_scanlines or jpeg_read_raw_data.
     */
    cinfo->global_state = cinfo->raw_data_out ? DSTATE_RAW_OK : DSTATE_SCANNING;
    return TRUE;
}

/*
 * Read some scanlines of data from the JPEG decompressor.
 *
 * The return value will be the number of lines actually read.
 * This may be less than the number requested in several cases,
 * including bottom of image, data source suspension, and operating
 * modes that emit multiple scanlines at a time.
 *
 * Note: we warn about excess calls to jpeg_read_scanlines() since
 * this likely signals an application programmer error. However,
 * an oversize buffer (max_lines > scanlines remaining) is not an error.
 */

GLOBAL(JDIMENSION)
jpeg_read_scanlines (j_decompress_ptr cinfo, JSAMPARRAY scanlines,
                    JDIMENSION max_lines)
{
    JDIMENSION row_ctr;

    if (cinfo->global_state != DSTATE_SCANNING)
        ERREXIT1(cinfo, JERR_BAD_STATE, cinfo->global_state);
    if (cinfo->output_scanline >= cinfo->output_height) {
        WARNMS(cinfo, JWRN_TOO_MUCH_DATA);
    }
}

```

```

/*
 * jdapistd.c
 *
 * Copyright (C) 1994-1996, Thomas G. Lane.
 * This file is part of the Independent JPEG Group's software.
 * For conditions of distribution and use, see the accompanying README file.
 *
 * This file contains application interface code for the decompression half
 * of the JPEG library. These are the "standard" API routines that are
 * used in the normal full-decompression case. They are not used by a
 * transcoding-only application. Note that if an application links in
 * jpeg_start_decompress, it will end up linking in the entire decompressor.
 * We thus must separate this file from jdapimin.c to avoid linking the
 * whole decompression library into a transcoder.
 */

#define JPEG_INTERNALS
#include "jinclude.h"
#include "jpeglib.h"

/* Forward declarations */
LOCAL(boolean) output_pass_setup JPP((j_decompress_ptr cinfo));

/*
 * Decompression initialization.
 * jpeg_read_header must be completed before calling this.
 *
 * If a multipass operating mode was selected, this will do all but the
 * last pass, and thus may take a great deal of time.
 *
 * Returns FALSE if suspended. The return value need be inspected only if
 * a suspending data source is used.
 */

GLOBAL(boolean)
jpeg_start_decompress (j_decompress_ptr cinfo)
{
  if (cinfo->global_state == DSTATE_READY) {
    /* First call: initialize master control, select active modules */
    jinit_master_decompress(cinfo);
    if (cinfo->buffered_image) {
      /* No more work here; expecting jpeg_start_output next */
      cinfo->global_state = DSTATE_BUFIMAGE;
      return TRUE;
    }
    cinfo->global_state = DSTATE_PRELOAD;
  }
  if (cinfo->global_state == DSTATE_PRELOAD) {
    /* If file has multiple scans, absorb them all into the coef buffer */
    if (cinfo->inputctl->has_multiple_scans) {
#ifdef D_MULTISCAN_FILES_SUPPORTED
      for (;;) {
        int retcode;
        /* Call progress monitor hook if present */
        if (cinfo->progress != NULL)
          (*cinfo->progress->progress_monitor) ((j_common_ptr) cinfo);
        /* Absorb some more input */
        retcode = (*cinfo->inputctl->consume_input) (cinfo);
        if (retcode == JPEG_SUSPENDED)
          return FALSE;
        if (retcode == JPEG_REACHED_EOI)
          break;
        /* Advance progress counter if appropriate */
        if (cinfo->progress != NULL &&
            (retcode == JPEG_ROW_COMPLETED || retcode == JPEG_REACHED_SOS)) {
          if (++cinfo->progress->pass_counter >= cinfo->progress->pass_limit) {
            /* jdmaster underestimated number of scans; ratchet up one scan */
            cinfo->progress->pass_limit += (long) cinfo->total_iMCU_rows;
          }
        }
      }
    }
  }
} else
  ERREXIT(cinfo, JERR_NOT_COMPILED);
#endif /* D_MULTISCAN_FILES_SUPPORTED */
}
cinfo->output_scan_number = cinfo->input_scan_number;
} else if (cinfo->global_state != DSTATE_PRESCAN)
  ERREXIT1(cinfo, JERR_BAD_STATE, cinfo->global_state);
}

```

```

    return 0;
}

/* Call progress monitor hook if present */
if (cinfo->progress != NULL) {
    cinfo->progress->pass_counter = (long) cinfo->output_scanline;
    cinfo->progress->pass_limit = (long) cinfo->output_height;
    (*cinfo->progress->progress_monitor) ((j_common_ptr) cinfo);
}

/* Process some data */
row_ctr = 0;
(*cinfo->main->process_data) (cinfo, scanlines, &row_ctr, max_lines);
cinfo->output_scanline += row_ctr;
return row_ctr;
}

/*
 * Alternate entry point to read raw data.
 * Processes exactly one iMCU row per call, unless suspended.
 */

GLOBAL(JDIMENSION)
jpeg_read_raw_data (j_decompress_ptr cinfo, JSAMPIMAGE data,
                    JDIMENSION max_lines)
{
    JDIMENSION lines_per_iMCU_row;

    if (cinfo->global_state != DSTATE_RAW_OK)
        ERREXIT1(cinfo, JERR_BAD_STATE, cinfo->global_state);
    if (cinfo->output_scanline >= cinfo->output_height) {
        WARNMS(cinfo, JWRN_TOO_MUCH_DATA);
        return 0;
    }

    /* Call progress monitor hook if present */
    if (cinfo->progress != NULL) {
        cinfo->progress->pass_counter = (long) cinfo->output_scanline;
        cinfo->progress->pass_limit = (long) cinfo->output_height;
        (*cinfo->progress->progress_monitor) ((j_common_ptr) cinfo);
    }

    /* Verify that at least one iMCU row can be returned. */
    lines_per_iMCU_row = cinfo->max_v_samp_factor * cinfo->min_DCT_scaled_size;
    if (max_lines < lines_per_iMCU_row)
        ERREXIT(cinfo, JERR_BUFFER_SIZE);

    /* Decompress directly into user's buffer. */
    if (! (*cinfo->coef->decompress_data) (cinfo, data))
        return 0; /* suspension forced, can do nothing more */

    /* OK, we processed one iMCU row. */
    cinfo->output_scanline += lines_per_iMCU_row;
    return lines_per_iMCU_row;
}

/* Additional entry points for buffered-image mode. */

#ifdef D_MULTISCAN_FILES_SUPPORTED

/*
 * Initialize for an output pass in buffered-image mode.
 */

GLOBAL(boolean)
jpeg_start_output (j_decompress_ptr cinfo, int scan_number)
{
    if (cinfo->global_state != DSTATE_BUFIMAGE &&
        cinfo->global_state != DSTATE_PRESCAN)
        ERREXIT1(cinfo, JERR_BAD_STATE, cinfo->global_state);
    /* Limit scan number to valid range */
    if (scan_number <= 0)
        scan_number = 1;
    if (cinfo->inputctl->eoi_reached &&
        scan_number > cinfo->input_scan_number)
        scan_number = cinfo->input_scan_number;
    cinfo->output_scan_number = scan_number;
    /* Perform any dummy output passes, and set up for the real pass */
}

```

```

    return output_pass_setup(cinfo);
}

/*
 * Finish up after an output pass in buffered-image mode.
 *
 * Returns FALSE if suspended. The return value need be inspected only if
 * a suspending data source is used.
 */

GLOBAL(boolean)
jpeg_finish_output (j_decompress_ptr cinfo)
{
    if ((cinfo->global_state == DSTATE_SCANNING ||
        cinfo->global_state == DSTATE_RAW_OK) && cinfo->buffered_image) {
        /* Terminate this pass. */
        /* We do not require the whole pass to have been completed. */
        (*cinfo->master->finish_output_pass) (cinfo);
        cinfo->global_state = DSTATE_BUFPOST;
    } else if (cinfo->global_state != DSTATE_BUFPOST) {
        /* BUFPOST = repeat call after a suspension, anything else is error */
        ERREXIT1(cinfo, JERR_BAD_STATE, cinfo->global_state);
    }
    /* Read markers looking for SOS or EOI */
    while (cinfo->input_scan_number <= cinfo->output_scan_number &&
        ! cinfo->inputctl->eoi_reached) {
        if ((*cinfo->inputctl->consume_input) (cinfo) == JPEG_SUSPENDED)
            return FALSE; /* Suspend, come back later */
    }
    cinfo->global_state = DSTATE_BUFIMAGE;
    return TRUE;
}
#endif /* D_MULTISCAN_FILES_SUPPORTED */

```

```

/*
 * jdatadst.c
 *
 * Copyright (C) 1994-1996, Thomas G. Lane.
 * This file is part of the Independent JPEG Group's software.
 * For conditions of distribution and use, see the accompanying README file.
 *
 * This file contains compression data destination routines for the case of
 * emitting JPEG data to a file (or any stdio stream). While these routines
 * are sufficient for most applications, some will want to use a different
 * destination manager.
 * IMPORTANT: we assume that fwrite() will correctly transcribe an array of
 * JOCTETs into 8-bit-wide elements on external storage. If char is wider
 * than 8 bits on your machine, you may need to do some tweaking.
 */

/* this is not a core library module, so it doesn't define JPEG_INTERNALS */
#include "jinclude.h"
#include "jpeglib.h"
#include "jerror.h"

/* ### a couple of new definitions to help differentiate between FILE and buffer I/O */
#define FILE_OUTPUT 1
#define BUFFER_OUTPUT 2

/* Expanded data destination object for stdio output */
typedef struct {
  struct jpeg_destination_mgr pub; /* public fields */

  FILE * outfile; /* target stream */
  JOCTET **outbuffer; /* /* target buffer */
  JOCTET * buffer; /* start of buffer */
  int n_BufferFlag; /* ### My addition. Flag determines whether to process FILE * or JOCTET * */
} my_destination_mgr;

typedef my_destination_mgr * my_dest_ptr;

#define OUTPUT_BUF_SIZE 4096 /* choose an efficiently fwrite'able size */

/* ### Imtiaz: Stitching routine to patch linked list into a single stream of JOCTET */
/* To store the stuff (JOCTET stream) in jpeg_destination_mgr.outbuffer; */
GLOBAL(void)
stitch_list(j_compress_ptr cinfo)
{
  int i;
  int chunk, count=0;
  buffer_list* list, *temp;

  list=cinfo->dest->head_ptr;
  chunk=(int)cinfo->index;

  /* Allocating memory. */

  cinfo->dest->outbuffer=(JOCTET *)malloc(cinfo->index*sizeof(JOCTET));
  if (cinfo->dest->outbuffer==NULL)
  {
    fprintf(stderr, "Memory Allocation Error\n");
    exit(1);
  }

  while (list->next!=NULL)
  {
    for(i=0;i<OUTPUT_BUF_SIZE;i++)
    {
      cinfo->dest->outbuffer[count]=list->buffer[i];
      count++;
    }

    temp=list;
    list=list->next;

    free(temp->buffer);
    free(temp);
  }
}

```

```

    chunk=chunk-(int)OUTPUT_BUF_SIZE;
}
/* the rest */
for(i=0;i<chunk;i++)
{
    cinfo->dest->outbuffer[count]=list->buffer[i];
    count++;
}

free(list->buffer);
free(list);
}

/*
 * Initialize destination --- called by jpeg_start_compress
 * before any data is actually written.
 */

METHODDEF(void)
init_destination (j_compress_ptr cinfo)
{
    my_dest_ptr dest = (my_dest_ptr) cinfo->dest;

    /* Allocate the output buffer --- it will be released when done with image */
    dest->buffer = (JOCTET *)
        (*cinfo->mem->alloc_small) ((j_common_ptr) cinfo, JPOOL_IMAGE,
            OUTPUT_BUF_SIZE * SIZEOF(JOCTET));

    dest->pub.next_output_byte = dest->buffer;
    dest->pub.free_in_buffer = OUTPUT_BUF_SIZE;
}

/*
 * Empty the output buffer --- called whenever buffer fills up.
 *
 * In typical applications, this should write the entire output buffer
 * (ignoring the current state of next_output_byte & free_in_buffer),
 * reset the pointer & count to the start of the buffer, and return TRUE
 * indicating that the buffer has been dumped.
 *
 * In applications that need to be able to suspend compression due to output
 * overrun, a FALSE return indicates that the buffer cannot be emptied now.
 * In this situation, the compressor will return to its caller (possibly with
 * an indication that it has not accepted all the supplied scanlines). The
 * application should resume compression after it has made more room in the
 * output buffer. Note that there are substantial restrictions on the use of
 * suspension --- see the documentation.
 *
 * When suspending, the compressor will back up to a convenient restart point
 * (typically the start of the current MCU). next_output_byte & free_in_buffer
 * indicate where the restart point will be if the current call returns FALSE.
 * Data beyond this point will be regenerated after resumption, so do not
 * write it out when emptying the buffer externally.
 */

METHODDEF(boolean)
empty_output_buffer (j_compress_ptr cinfo)
{
    int i, index, increment;
    buffer_list *bl;

    my_dest_ptr dest = (my_dest_ptr) cinfo->dest;

    if (dest->n_BufferFlag==FILE_OUTPUT)
    {
        if (JFWRITE(dest->outfile, dest->buffer, OUTPUT_BUF_SIZE) !=
            (size_t) OUTPUT_BUF_SIZE)
            ERREXIT(cinfo, JERR_FILE_WRITE);
    }
    if (dest->n_BufferFlag==BUFFER_OUTPUT)
    {
        /* ### Imtiaz: First thing we know is that this part deals with a fixed length dump. */

```



```

bl=(buffer_list *)malloc(sizeof(buffer_list));
if (bl==NULL)
{
    fprintf(stderr,"Memory Allocation Error\n");
    exit(1);
}

bl->buffer=(JOCTET *)malloc(OUTPUT_BUF_SIZE*sizeof(JOCTET));

bl->next=NULL;

/* For the first time we need to set the head_ptr and current_ptr in jpeg_destination_mgr accordingl
y. Yes! I changed that too. */
if (cinfo->dest->head_ptr==NULL)
{
    cinfo->dest->current_ptr=bl;
    cinfo->dest->head_ptr=cinfo->dest->current_ptr;
}
else
{
    cinfo->dest->current_ptr->next=bl;
    cinfo->dest->current_ptr=bl;
}

index=cinfo->index;

increment=OUTPUT_BUF_SIZE;

for (i=0;i<OUTPUT_BUF_SIZE;i++)
    cinfo->dest->current_ptr->buffer[i]=dest->buffer[i];

cinfo->index=index+(int)OUTPUT_BUF_SIZE;

dest->pub.next_output_byte = dest->buffer;
dest->pub.free_in_buffer = OUTPUT_BUF_SIZE;

return TRUE;
}

/* Terminate destination --- called by jpeg_finish_compress
after all data has been written. Usually needs to flush buffer.

NB: *not* called by jpeg_abort or jpeg_destroy; surrounding
application must deal with any cleanup that should happen even
for error exit.
*/

METHODDEF(void)
term_destination (j_compress_ptr cinfo)
{
    int i,index;
    buffer_list *bl;

    my_dest_ptr dest = (my_dest_ptr) cinfo->dest;
    size_t datacount = OUTPUT_BUF_SIZE - dest->pub.free_in_buffer;

    /* Write any data remaining in the buffer */
    if (datacount > 0) {
        if (dest->n_BufferFlag==FILE_OUTPUT)
        {
            if (JFWRITE(dest->outfile, dest->buffer, datacount) != datacount)
                ERREXIT(cinfo, JERR_FILE_WRITE);

            fflush(dest->outfile);

            /* Make sure we wrote the output file OK */
            if (ferror(dest->outfile))
                ERREXIT(cinfo, JERR_FILE_WRITE);
        }
        if (dest->n_BufferFlag==BUFFER_OUTPUT)
        {

```

```

/* ### Imtiaz: First thing we know is that this part deals with a fixed length dump. */

bl=(buffer_list *)malloc(sizeof(buffer_list));
if (bl==NULL)
{
    fprintf(stderr, "Memory Allocation Error\n");
    exit(1);
}

bl->buffer=(JOCTET *)malloc(datacount*sizeof(JOCTET));

bl->next=NULL;

/* For the first time we need to set the head_ptr and current_ptr in jpeg_destination_mgr accordingly. Yes! I changed that too. */
if (cinfo->dest->head_ptr==NULL)
{
    cinfo->dest->current_ptr=bl;
    cinfo->dest->head_ptr=cinfo->dest->current_ptr;
}
else
{
    cinfo->dest->current_ptr->next=bl;
    cinfo->dest->current_ptr=bl;
}

index=cinfo->index;

for (i=0;i<(int)datacount;i++)
    cinfo->dest->current_ptr->buffer[i]=dest->buffer[i];

cinfo->index=index+datacount;

/* The end... Now to stitch. :) */
stitch_list(cinfo);
}

/*
 * Prepare for output to a stdio stream.
 * The caller must have already opened the stream, and is responsible
 * for closing it after finishing compression.
 */

GLOBAL(void)
jpeg_stdio_dest (j_compress_ptr cinfo, FILE * outfile)
{
    my_dest_ptr dest;

    /* The destination object is made permanent so that multiple JPEG images
     * can be written to the same file without re-executing jpeg_stdio_dest.
     * This makes it dangerous to use this manager and a different destination
     * manager serially with the same JPEG object, because their private object
     * sizes may be different.  Caveat programmer.
     */
    if (cinfo->dest == NULL) { /* first time for this JPEG object? */
        cinfo->dest = (struct jpeg_destination_mgr *)
            (*cinfo->mem->alloc_small) ((j_common_ptr) cinfo, JPOOL_PERMANENT,
                SIZEOF(my_destination_mgr));
    }

    dest = (my_dest_ptr) cinfo->dest;

    /* ### Setting flag to process FILE output */
    dest->n_BufferFlag=FILE_OUTPUT;

    dest->pub.init_destination = init_destination;
    dest->pub.empty_output_buffer = empty_output_buffer;
    dest->pub.term_destination = term_destination;
    dest->outfile = outfile;
}

```

```

}

GLOBAL(void)
jpeg_buffer_dest (j_compress_ptr cinfo)
{
    my_dest_ptr dest;

    /* The destination object is made permanent so that multiple JPEG images
     * can be written to the same file without re-executing jpeg_stdio_dest.
     * This makes it dangerous to use this manager and a different destination
     * manager serially with the same JPEG object, because their private object
     * sizes may be different.  Caveat programmer.
     */
    if (cinfo->dest == NULL) { /* first time for this JPEG object? */
        cinfo->dest = (struct jpeg_destination_mgr *)
            (*cinfo->mem->alloc_small) ((j_common_ptr) cinfo, JPOOL_PERMANENT,
                sizeof(my_destination_mgr));
    }

    dest = (my_dest_ptr) cinfo->dest;

    /* ### Imtiaz: Setting flag to process BUFFER output */
    dest->n_BufferFlag=BUFFER_OUTPUT;

    dest->pub.init_destination = init_destination;
    dest->pub.empty_output_buffer = empty_output_buffer;
    dest->pub.term_destination = term_destination;
    /* ### cinfo->dest->outbuffer = buff; */

    cinfo->dest->head_ptr=NULL;
    cinfo->dest->current_ptr=NULL;
}

```

```

/*
 * jdatasrc.c
 *
 * Copyright (C) 1994-1996, Thomas G. Lane.
 * This file is part of the Independent JPEG Group's software.
 * For conditions of distribution and use, see the accompanying README file.
 *
 * This file contains decompression data source routines for the case of
 * reading JPEG data from a file (or any stdio stream). While these routines
 * are sufficient for most applications, some will want to use a different
 * source manager.
 * IMPORTANT: we assume that fread() will correctly transcribe an array of
 * JOCTETs from 8-bit-wide elements on external storage. If char is wider
 * than 8 bits on your machine, you may need to do some tweaking.
 */

/* this is not a core library module, so it doesn't define JPEG_INTERNALS */
#include "jinclude.h"
#include "jpeglib.h"
#include "jerror.h"

#define FILE_OUTPUT      1    /* ### Imtiaz: I put this here. */
#define BUFFER_OUTPUT    2

/* Expanded data source object for stdio input */
typedef struct {
  struct jpeg_source_mgr pub; /* public fields */

  FILE * infile;             /* source stream */
  JOCTET * buffer;           /* start of buffer */
  boolean start_of_file;     /* have we gotten any data yet? */
  long count_track;

  int n_BufferFlag;          /* ### Imtiaz: My addition. Flag determines whether to process FILE * or JOCTET * */
} my_source_mgr;

typedef my_source_mgr * my_src_ptr;

#define INPUT_BUF_SIZE 4096 /* choose an efficiently fread'able size */

/*
 * Initialize source --- called by jpeg_read_header
 * before any data is actually read.
 */
METHODDEF(void)
init_source (j_decompress_ptr cinfo)
{
  my_src_ptr src = (my_src_ptr) cinfo->src;

  /* We reset the empty-input-file flag for each image,
   * but we don't clear the input buffer.
   * This is correct behavior for reading a series of images from one source.
   */
  src->start_of_file = TRUE;
}

/*
 * Fill the input buffer --- called whenever buffer is emptied.
 *
 * In typical applications, this should read fresh data into the buffer
 * (ignoring the current state of next_input_byte & bytes_in_buffer),
 * reset the pointer & count to the start of the buffer, and return TRUE
 * indicating that the buffer has been reloaded. It is not necessary to
 * fill the buffer entirely, only to obtain at least one more byte.
 *
 * There is no such thing as an EOF return. If the end of the file has been
 * reached, the routine has a choice of ERREXIT() or inserting fake data into
 * the buffer. In most cases, generating a warning message and inserting a
 * fake EOI marker is the best course of action --- this will allow the
 * decompressor to output however much of the image is there. However,
 * the resulting error message is misleading if the real problem is an empty
 * input file, so we handle that case specially.
 */

```

```

* In applications that need to be able to suspend compression due to input
* not being available yet, a FALSE return indicates that no more data can be
* obtained right now, but more may be forthcoming later. In this situation,
* the decompressor will return to its caller (with an indication of the
* number of scanlines it has read, if any). The application should resume
* decompression after it has loaded more data into the input buffer. Note
* that there are substantial restrictions on the use of suspension --- see
* the documentation.
*
* When suspending, the decompressor will back up to a convenient restart point
* (typically the start of the current MCU). next_input_byte & bytes_in_buffer
* indicate where the restart point will be if the current call returns FALSE.
* Data beyond this point must be rescanned after resumption, so move it to
* the front of the buffer rather than discarding it.
*/

```

```

METHODDEF(boolean)
fill_input_buffer (j_decompress_ptr cinfo)
{
    long count;
    static int p;

    my_src_ptr src = (my_src_ptr) cinfo->src;
    size_t nbytes;

    if (src->n_BufferFlag==FILE_OUTPUT)
    {
        nbytes = JFREAD(src->infile, src->buffer, INPUT_BUF_SIZE);
    }
    else
    {
        count=0;
        while ((p<cinfo->src->buffer_length)&&(count<INPUT_BUF_SIZE))
        {
            src->buffer[count]= cinfo->src->inbuffer[0];
            count++;

            cinfo->src->inbuffer++;
            p++;
        }
    }

    nbytes=(size_t)count;

    if (nbytes <= 0) {
        if (src->n_BufferFlag==FILE_OUTPUT)
        {
            if (src->start_of_file) /* Treat empty input file as fatal error */
                ERREXIT(cinfo, JERR_INPUT_EMPTY);
            WARNMS(cinfo, JWRN_JPEG_EOF);
        }
        else
        { /* if dest->nBufferFlag==BUFFER_OUTPUT */
            fprintf(stderr, "Buffer Empty\n");
            fprintf(stderr, "Using Fake EOI marker.. \n");
        }
        /* Insert a fake EOI marker */
        src->buffer[0] = (JOCTET) 0xFF;
        src->buffer[1] = (JOCTET) JPEG_EOI;
        nbytes = 2;
    }

    src->pub.next_input_byte = src->buffer;
    src->pub.bytes_in_buffer = nbytes;
    src->start_of_file = FALSE;

    return TRUE;
}

/*
* Skip data --- used to skip over a potentially large amount of
* uninteresting data (such as an APPn marker).
*
* Writers of suspendable-input applications must note that skip_input_data
* is not granted the right to give a suspension return. If the skip extends

```

```

* beyond the data currently in the buffer, the buffer can be marked empty so
* that the next read will cause a fill_input_buffer call that can depend.
* Arranging for additional bytes to be discarded before reloading the input
* buffer is the application writer's problem.
*/

```

```

METHODDEF(void)
skip_input_data (j_decompress_ptr cinfo, long num_bytes)
{
    my_src_ptr src = (my_src_ptr) cinfo->src;

    /* Just a dumb implementation for now. Could use fseek() except
     * it doesn't work on pipes. Not clear that being smart is worth
     * any trouble anyway --- large skips are infrequent.
     */
    if (num_bytes > 0) {
        while (num_bytes > (long) src->pub.bytes_in_buffer) {
            num_bytes -= (long) src->pub.bytes_in_buffer;
            (void) fill_input_buffer(cinfo);
            /* note we assume that fill_input_buffer will never return FALSE,
             * so suspension need not be handled.
             */
        }
        src->pub.next_input_byte += (size_t) num_bytes;
        src->pub.bytes_in_buffer -= (size_t) num_bytes;
    }
}

```

```

/*
 * An additional method that can be provided by data source modules is the
 * resync_to_restart method for error recovery in the presence of RST markers.
 * For the moment, this source module just uses the default resync method
 * provided by the JPEG library. That method assumes that no backtracking
 * is possible.
 */

```

```

/*
 * Terminate source --- called by jpeg_finish_decompress
 * after all data has been read. Often a no-op.
 *
 * NB: *not* called by jpeg_abort or jpeg_destroy; surrounding
 * application must deal with any cleanup that should happen even
 * for error exit.
 */

```

```

METHODDEF(void)
term_source (j_decompress_ptr cinfo)
{
    /* no work necessary here */
}

```

```

/*
 * Prepare for input from a stdio stream.
 * The caller must have already opened the stream, and is responsible
 * for closing it after finishing decompression.
 */

```

```

GLOBAL(void)
jpeg_stdio_src (j_decompress_ptr cinfo, FILE * infile)
{
    my_src_ptr src;

    /* The source object and input buffer are made permanent so that a series
     * of JPEG images can be read from the same file by calling jpeg_stdio_src
     * only before the first one. (If we discarded the buffer at the end of
     * one image, we'd likely lose the start of the next one.)
     * This makes it unsafe to use this manager and a different source
     * manager serially with the same JPEG object. Caveat programmer.
     */
    if (cinfo->src == NULL) { /* first time for this JPEG object? */
        cinfo->src = (struct jpeg_source_mgr *)
            (*cinfo->mem->alloc_small) ((j_common_ptr) cinfo, JPOOL_PERMANENT,
                                      SIZEOF(my_source_mgr));
        src = (my_src_ptr) cinfo->src;
        src->buffer = (JOCTET *)
            (*cinfo->mem->alloc_small) ((j_common_ptr) cinfo, JPOOL_PERMANENT,
                                      INPUT_BUF_SIZE * SIZEOF(JOCTET));
    }
}

```

```

    }

    src = (my_src_ptr) cinfo->src;
    src->pub.init_source = init_source;
    src->pub.fill_input_buffer = fill_input_buffer;
    src->pub.skip_input_data = skip_input_data;
    src->pub.resync_to_restart = jpeg_resync_to_restart; /* use default method */
    src->pub.term_source = term_source;
    src->infile = infile;
    src->pub.bytes_in_buffer = 0; /* forces fill_input_buffer on first read */
    src->pub.next_input_byte = NULL; /* until buffer loaded */
}

```

```

GLOBAL(void)
jpeg_buffer_src (j_decompress_ptr cinfo)
{
    my_src_ptr src;

    /* The source object and input buffer are made permanent so that a series
     * of JPEG images can be read from the same file by calling jpeg_stdio_src
     * only before the first one. (If we discarded the buffer at the end of
     * one image, we'd likely lose the start of the next one.)
     * This makes it unsafe to use this manager and a different source
     * manager serially with the same JPEG object. Caveat programmer.
     */
    if (cinfo->src == NULL) { /* first time for this JPEG object? */
        cinfo->src = (struct jpeg_source_mgr *)
            (*cinfo->mem->alloc_small) ((j_common_ptr) cinfo, JPOOL_PERMANENT,
                                      SIZEOF(my_source_mgr));
        src = (my_src_ptr) cinfo->src;
        src->buffer = (JOCTET *)
            (*cinfo->mem->alloc_small) ((j_common_ptr) cinfo, JPOOL_PERMANENT,
                                      INPUT_BUF_SIZE * SIZEOF(JOCTET));

        src = (my_src_ptr) cinfo->src;

        /* ### Imtiaz: Setting flag to process BUFFER output */
        src->n_BufferFlag=BUFFER_OUTPUT;

        src->pub.init_source = init_source;
        src->pub.fill_input_buffer = fill_input_buffer;
        src->pub.skip_input_data = skip_input_data;
        src->pub.resync_to_restart = jpeg_resync_to_restart; /* use default method */
        src->pub.term_source = term_source;

        src->pub.bytes_in_buffer = 0; /* forces fill_input_buffer on first read */
        src->pub.next_input_byte = NULL; /* until buffer loaded */
    }
}

```

```

/*
 * jdcoefct.c
 *
 * Copyright (C) 1994-1997, Thomas G. Lane.
 * This file is part of the Independent JPEG Group's software.
 * For conditions of distribution and use, see the accompanying README file.
 *
 * This file contains the coefficient buffer controller for decompression.
 * This controller is the top level of the JPEG decompressor proper.
 * The coefficient buffer lies between entropy decoding and inverse-DCT steps.
 *
 * In buffered-image mode, this controller is the interface between
 * input-oriented processing and output-oriented processing.
 * Also, the input side (only) is used when reading a file for transcoding.
 */

#define JPEG_INTERNALS
#include "jinclude.h"
#include "jpeglib.h"

/* Block smoothing is only applicable for progressive JPEG, so: */
#ifdef D_PROGRESSIVE_SUPPORTED
#undef BLOCK_SMOOTHING_SUPPORTED
#endif

/* Private buffer controller object */
typedef struct {
  struct jpeg_d_coef_controller pub; /* public fields */

  /* These variables keep track of the current location of the input side. */
  /* cinfo->input_iMCU_row is also used for this. */
  int DIMENSION MCU_ctr; /* counts MCUs processed in current row */
  int MCU_vert_offset; /* counts MCU rows within iMCU row */
  int MCU_rows_per_iMCU_row; /* number of such rows needed */

  /* The output side's location is represented by cinfo->output_iMCU_row. */

  /* In single-pass modes, it's sufficient to buffer just one MCU.
   * We allocate a workspace of D_MAX_BLOCKS_IN_MCU coefficient blocks,
   * and let the entropy decoder write into that workspace each time.
   * (On 80x86, the workspace is FAR even though it's not really very big;
   * this is to keep the module interfaces unchanged when a large coefficient
   * buffer is necessary.)
   * In multi-pass modes, this array points to the current MCU's blocks
   * within the virtual arrays; it is used only by the input side.
   */
  JBLOCKROW MCU_buffer[D_MAX_BLOCKS_IN_MCU];

#ifdef D_MULTISCAN_FILES_SUPPORTED
  /* In multi-pass modes, we need a virtual block array for each component. */
  jvirt_barray_ptr whole_image[MAX_COMPONENTS];
#endif

#ifdef BLOCK_SMOOTHING_SUPPORTED
  /* When doing block smoothing, we latch coefficient Al values here */
  int * coef_bits_latch;
#define SAVED_COEFS 6 /* we save coef_bits[0..5] */
#endif
} my_coef_controller;

typedef my_coef_controller * my_coef_ptr;

/* Forward declarations */
METHODDEF(int) decompress_onepass
    (JPP((j_decompress_ptr cinfo, JSAMPIMAGE output_buf)));
#ifdef D_MULTISCAN_FILES_SUPPORTED
METHODDEF(int) decompress_data
    (JPP((j_decompress_ptr cinfo, JSAMPIMAGE output_buf)));
#endif
#ifdef BLOCK_SMOOTHING_SUPPORTED
LOCAL(boolean) smoothing_ok(JPP((j_decompress_ptr cinfo)));
METHODDEF(int) decompress_smooth_data
    (JPP((j_decompress_ptr cinfo, JSAMPIMAGE output_buf)));
#endif

LOCAL(void)
start_iMCU_row(j_decompress_ptr cinfo)
/* Reset within-iMCU-row counters for a new row (input side) */

```



```

{
    my_coef_ptr coef = (my_coef_ptr) cinfo->coef;

    /* In an interleaved scan, an MCU row is the same as an iMCU row.
     * In a noninterleaved scan, an iMCU row has v_samp_factor MCU rows.
     * But at the bottom of the image, process only what's left.
     */
    if (cinfo->comps_in_scan > 1) {
        coef->MCU_rows_per_iMCU_row = 1;
    } else {
        if (cinfo->input_iMCU_row < (cinfo->total_iMCU_rows-1))
            coef->MCU_rows_per_iMCU_row = cinfo->cur_comp_info[0]->v_samp_factor;
        else
            coef->MCU_rows_per_iMCU_row = cinfo->cur_comp_info[0]->last_row_height;
    }

    coef->MCU_ctr = 0;
    coef->MCU_vert_offset = 0;
}

/*
 * Initialize for an input processing pass.
 */

METHODDEF(void)
start_input_pass (j_decompress_ptr cinfo)
{
    cinfo->input_iMCU_row = 0;
    start_iMCU_row(cinfo);
}

/*
 * Initialize for an output processing pass.
 */

METHODDEF(void)
start_output_pass (j_decompress_ptr cinfo)
{
#ifdef BLOCK_SMOOTHING_SUPPORTED
    my_coef_ptr coef = (my_coef_ptr) cinfo->coef;

    /* If multipass, check to see whether to use block smoothing on this pass */
    if (coef->pub.coef_arrays != NULL) {
        if (cinfo->do_block_smoothing && smoothing_ok(cinfo))
            coef->pub.decompress_data = decompress_smooth_data;
        else
            coef->pub.decompress_data = decompress_data;
    }
#endif
    cinfo->output_iMCU_row = 0;
}

/*
 * Decompress and return some data in the single-pass case.
 * Always attempts to emit one fully interleaved MCU row ("iMCU" row).
 * Input and output must run in lockstep since we have only a one-MCU buffer.
 * Return value is JPEG_ROW_COMPLETED, JPEG_SCAN_COMPLETED, or JPEG_SUSPENDED.
 *
 * NB: output_buf contains a plane for each component in image,
 * which we index according to the component's SOF position.
 */

METHODDEF(int)
decompress_onepass (j_decompress_ptr cinfo, JSAMPIMAGE output_buf)
{
    my_coef_ptr coef = (my_coef_ptr) cinfo->coef;
    JDIMENSION MCU_col_num; /* index of current MCU within row */
    JDIMENSION last_MCU_col = cinfo->MCUs_per_row - 1;
    JDIMENSION last_iMCU_row = cinfo->total_iMCU_rows - 1;
    int blkn, ci, xindex, yindex, yoffset, useful_width;
    JSAMPARRAY output_ptr;
    JDIMENSION start_col, output_col;
    jpeg_component_info *comp_ptr;
    inverse_DCT_method_ptr inverse_DCT;

    /* Loop to process as much as one whole iMCU row */
    for (yoffset = coef->MCU_vert_offset; yoffset < coef->MCU_rows_per_iMCU_row;

```

```

        yoffset++) {
    for (MCU_col_num = coef->MCU_col_ctr; MCU_col_num <= last_MCU_col;
        MCU_col_num++) {
        /* Try to fetch an MCU. Entropy decoder expects buffer to be zeroed. */
        jzero_far((void *) coef->MCU_buffer[0],
            (size_t) (cinfo->blocks_in_MCU * SIZEOF(JBLOCK)));
        if (! (*cinfo->entropy->decode_mcu) (cinfo, coef->MCU_buffer)) {
            /* Suspension forced; update state counters and exit */
            coef->MCU_vert_offset = yoffset;
            coef->MCU_ctr = MCU_col_num;
            return JPEG_SUSPENDED;
        }
        /* Determine where data should go in output_buf and do the IDCT thing.
         * We skip dummy blocks at the right and bottom edges (but blk_n gets
         * incremented past them!). Note the inner loop relies on having
         * allocated the MCU_buffer[] blocks sequentially.
         */
        blk_n = 0; /* index of current DCT block within MCU */
        for (ci = 0; ci < cinfo->comps_in_scan; ci++) {
            comp_ptr = cinfo->cur_comp_info[ci];
            /* Don't bother to IDCT an uninteresting component. */
            if (! comp_ptr->component_needed) {
                blk_n += comp_ptr->MCU_blocks;
                continue;
            }
            inverse_DCT = cinfo->idct->inverse_DCT[comp_ptr->component_index];
            useful_width = (MCU_col_num < last_MCU_col) ? comp_ptr->MCU_width
                : comp_ptr->last_col_width;
            output_ptr = output_buf[comp_ptr->component_index] +
                yoffset * comp_ptr->DCT_scaled_size;
            start_col = MCU_col_num * comp_ptr->MCU_sample_width;
            for (yindex = 0; yindex < comp_ptr->MCU_height; yindex++) {
                if (cinfo->input_iMCU_row < last_iMCU_row ||
                    yoffset+yindex < comp_ptr->last_row_height) {
                    output_col = start_col;
                    for (xindex = 0; xindex < useful_width; xindex++) {
                        (*inverse_DCT) (cinfo, comp_ptr,
                            (JCOEFPTR) coef->MCU_buffer[blk_n+xindex],
                            output_ptr, output_col);
                        output_col += comp_ptr->DCT_scaled_size;
                    }
                    blk_n += comp_ptr->MCU_width;
                    output_ptr += comp_ptr->DCT_scaled_size;
                }
            }
            /* Completed an MCU row, but perhaps not an iMCU row */
            coef->MCU_ctr = 0;
        }
        /* Completed the iMCU row, advance counters for next one */
        cinfo->output_iMCU_row++;
        if (++(cinfo->input_iMCU_row) < cinfo->total_iMCU_rows) {
            start_iMCU_row(cinfo);
            return JPEG_ROW_COMPLETED;
        }
        /* Completed the scan */
        (*cinfo->inputctl->finish_input_pass) (cinfo);
        return JPEG_SCAN_COMPLETED;
    }
}

/*
 * Dummy consume-input routine for single-pass operation.
 */

METHODDEF(int)
dummy_consume_data (j_decompress_ptr cinfo)
{
    return JPEG_SUSPENDED; /* Always indicate nothing was done */
}

#ifdef D_MULTISCAN_FILES_SUPPORTED

/*
 * Consume input data and store it in the full-image coefficient buffer.
 * We read as much as one fully interleaved MCU row ("iMCU" row) per call,
 * ie, v_samp_factor block rows for each component in the scan.
 * Return value is JPEG_ROW_COMPLETED, JPEG_SCAN_COMPLETED, or JPEG_SUSPENDED.
 */

```

```

*/
METHODDEF(int)
consume_data (j_decompress_ptr cinfo)
{
    my_coef_ptr coef = (my_coef_ptr) cinfo->coef;
    JDIMENSION MCU_col_num; /* index of current MCU within row */
    int blkn, ci, xindex, yindex, yoffset;
    JDIMENSION start_col;
    JBLOCKARRAY buffer[MAX_COMPS_IN_SCAN];
    JBLOCKROW buffer_ptr;
    jpeg_component_info *comp_ptr;

    /* Align the virtual buffers for the components used in this scan. */
    for (ci = 0; ci < cinfo->comps_in_scan; ci++) {
        comp_ptr = cinfo->cur_comp_info[ci];
        buffer[ci] = (*cinfo->mem->access_virt_barray)
            ((j_common_ptr) cinfo, coef->whole_image[comp_ptr->component_index],
             cinfo->input_iMCU_row * comp_ptr->v_samp_factor,
             (JDIMENSION) comp_ptr->v_samp_factor, TRUE);
        /* Note: entropy decoder expects buffer to be zeroed,
         * but this is handled automatically by the memory manager
         * because we requested a pre-zeroed array.
         */
    }

    /* Loop to process one whole iMCU row */
    for (yoffset = coef->MCU_vert_offset; yoffset < coef->MCU_rows_per_iMCU_row;
         yoffset++) {
        for (MCU_col_num = coef->MCU_ctr; MCU_col_num < cinfo->MCUs_per_row;
             MCU_col_num++) {
            /* Construct list of pointers to DCT blocks belonging to this MCU */
            blkn = 0; /* index of current DCT block within MCU */
            for (ci = 0; ci < cinfo->comps_in_scan; ci++) {
                comp_ptr = cinfo->cur_comp_info[ci];
                start_col = MCU_col_num * comp_ptr->MCU_width;
                for (yindex = 0; yindex < comp_ptr->MCU_height; yindex++) {
                    buffer_ptr = buffer[ci][yindex+yoffset] + start_col;
                    for (xindex = 0; xindex < comp_ptr->MCU_width; xindex++) {
                        coef->MCU_buffer[blkn++] = buffer_ptr++;
                    }
                }
            }
            /* Try to fetch the MCU. */
            if (!(*cinfo->entropy->decode_mcu) (cinfo, coef->MCU_buffer)) {
                /* Suspension forced; update state counters and exit */
                coef->MCU_vert_offset = yoffset;
                coef->MCU_ctr = MCU_col_num;
                return JPEG_SUSPENDED;
            }
        }
        /* Completed an MCU row, but perhaps not an iMCU row */
        coef->MCU_ctr = 0;
    }
    /* Completed the iMCU row, advance counters for next one */
    if (++(cinfo->input_iMCU_row) < cinfo->total_iMCU_rows) {
        start_iMCU_row(cinfo);
        return JPEG_ROW_COMPLETED;
    }
    /* Completed the scan */
    (*cinfo->inputctl->finish_input_pass) (cinfo);
    return JPEG_SCAN_COMPLETED;
}

/*
 * Decompress and return some data in the multi-pass case.
 * Always attempts to emit one fully interleaved MCU row ("iMCU" row).
 * Return value is JPEG_ROW_COMPLETED, JPEG_SCAN_COMPLETED, or JPEG_SUSPENDED.
 * NB: output_buf contains a plane for each component in image.
 */

```

```

METHODDEF(int)
decompress_data (j_decompress_ptr cinfo, JSAMPIMAGE output_buf)
{
    my_coef_ptr coef = (my_coef_ptr) cinfo->coef;
    JDIMENSION last_iMCU_row = cinfo->total_iMCU_rows - 1;
    JDIMENSION block_num;
    int ci, block_row, block_rows;

```

```

JBLOCKARRAY buffer;
JBLOCKROW buffer_ptr;
JSAMPARRAY output_ptr;
JDIMENSION output_col;
jpeg_component_info *comp_ptr;
inverse_DCT_method_ptr inverse_DCT;

/* Force some input to be done if we are getting ahead of the input. */
while (cinfo->input_scan_number < cinfo->output_scan_number ||
      (cinfo->input_scan_number == cinfo->output_scan_number &&
       cinfo->input_iMCU_row <= cinfo->output_iMCU_row)) {
    if ((*cinfo->inputctl->consume_input)(cinfo) == JPEG_SUSPENDED)
        return JPEG_SUSPENDED;
}

/* OK, output from the virtual arrays. */
for (ci = 0, comp_ptr = cinfo->comp_info; ci < cinfo->num_components;
     ci++, comp_ptr++) {
    /* Don't bother to IDCT an uninteresting component. */
    if (!comp_ptr->component_needed)
        continue;
    /* Align the virtual buffer for this component. */
    buffer = (*cinfo->mem->access_virt_barray)
        ((j_common_ptr) cinfo, coef->whole_image[ci],
         cinfo->output_iMCU_row * comp_ptr->v_samp_factor,
         (JDIMENSION) comp_ptr->v_samp_factor, FALSE);
    /* Count non-dummy DCT block rows in this iMCU row. */
    if (cinfo->output_iMCU_row < last_iMCU_row)
        block_rows = comp_ptr->v_samp_factor;
    else {
        /* NB: can't use last_row_height here; it is input-side-dependent! */
        block_rows = (int) (comp_ptr->height_in_blocks % comp_ptr->v_samp_factor);
        if (block_rows == 0) block_rows = comp_ptr->v_samp_factor;
    }
    inverse_DCT = cinfo->idct->inverse_DCT[ci];
    output_ptr = output_buf[ci];
    /* Loop over all DCT blocks to be processed. */
    for (block_row = 0; block_row < block_rows; block_row++) {
        buffer_ptr = buffer[block_row];
        output_col = 0;
        for (block_num = 0; block_num < comp_ptr->width_in_blocks; block_num++) {
            (*inverse_DCT) (cinfo, comp_ptr, (JCOEFPTR) buffer_ptr,
                           output_ptr, output_col);
            buffer_ptr++;
            output_col += comp_ptr->DCT_scaled_size;
        }
        output_ptr += comp_ptr->DCT_scaled_size;
    }
    if (++(cinfo->output_iMCU_row) < cinfo->total_iMCU_rows)
        return JPEG_ROW_COMPLETED;
    return JPEG_SCAN_COMPLETED;
}

#endif /* D_MULTISCAN_FILES_SUPPORTED */

#ifdef BLOCK_SMOOTHING_SUPPORTED

/*
 * This code applies interblock smoothing as described by section K.8
 * of the JPEG standard: the first 5 AC coefficients are estimated from
 * the DC values of a DCT block and its 8 neighboring blocks.
 * We apply smoothing only for progressive JPEG decoding, and only if
 * the coefficients it can estimate are not yet known to full precision.
 */

/* Natural-order array positions of the first 5 zigzag-order coefficients */
#define Q01_POS 1
#define Q10_POS 8
#define Q20_POS 16
#define Q11_POS 9
#define Q02_POS 2

/*
 * Determine whether block smoothing is applicable and safe.
 * We also latch the current states of the coef_bits[] entries for the
 * AC coefficients; otherwise, if the input side of the decompressor
 * advances into a new scan, we might think the coefficients are known

```

```

* more accurately than they really are.
*/

```

```

LOCAL(boolean)
smoothing_ok (j_decompress_ptr cinfo)
{
    my_coef_ptr coef = (my_coef_ptr) cinfo->coef;
    boolean smoothing_useful = FALSE;
    int ci, coefi;
    jpeg_component_info *comp_ptr;
    JQUANT_TBL * qtable;
    int * coef_bits;
    int * coef_bits_latch;

    if (! cinfo->progressive_mode || cinfo->coef_bits == NULL)
        return FALSE;

    /* Allocate latch area if not already done */
    if (coef->coef_bits_latch == NULL)
        coef->coef_bits_latch = (int *)
            (*cinfo->mem->alloc_small) ((j_common_ptr) cinfo, JPOOL_IMAGE,
                cinfo->num_components *
                (SAVED_COEFS * SIZEOF(int)));
    coef_bits_latch = coef->coef_bits_latch;

    for (ci = 0, comp_ptr = cinfo->comp_info; ci < cinfo->num_components;
        ci++, comp_ptr++) {
        /* All components' quantization values must already be latched. */
        if ((qtable = comp_ptr->quant_table) == NULL)
            return FALSE;
        /* Verify DC & first 5 AC quantizers are nonzero to avoid zero-divide. */
        if (qtable->quantval[0] == 0 ||
            qtable->quantval[Q01_POS] == 0 ||
            qtable->quantval[Q10_POS] == 0 ||
            qtable->quantval[Q20_POS] == 0 ||
            qtable->quantval[Q11_POS] == 0 ||
            qtable->quantval[Q02_POS] == 0)
            return FALSE;
        /* DC values must be at least partly known for all components. */
        coef_bits = cinfo->coef_bits[ci];
        if (coef_bits[0] < 0)
            return FALSE;
        /* Block smoothing is helpful if some AC coefficients remain inaccurate. */
        for (coefi = 1; coefi <= 5; coefi++) {
            coef_bits_latch[coefi] = coef_bits[coefi];
            if (coef_bits[coefi] != 0)
                smoothing_useful = TRUE;
        }
        coef_bits_latch += SAVED_COEFS;
    }
    return smoothing_useful;
}

/*
* Variant of decompress_data for use when doing block smoothing.
*/

```

```

METHODDEF(int)
decompress_smooth_data (j_decompress_ptr cinfo, JSAMPIMAGE output_buf)
{
    my_coef_ptr coef = (my_coef_ptr) cinfo->coef;
    JDIMENSION last_iMCU_row = cinfo->total_iMCU_rows - 1;
    JDIMENSION block_num, last_block_column;
    int ci, block_row, block_rows, access_rows;
    JBLOCKARRAY buffer;
    JBLOCKROW buffer_ptr, prev_block_row, next_block_row;
    JSAMPARRAY output_ptr;
    JDIMENSION output_col;
    jpeg_component_info *comp_ptr;
    inverse_DCT_method_ptr inverse_DCT;
    boolean first_row, last_row;
    JBLOCK workspace;
    int *coef_bits;
    JQUANT_TBL *quanttbl;
    INT32 Q00,Q01,Q02,Q10,Q11,Q20, num;
    int DC1,DC2,DC3,DC4,DC5,DC6,DC7,DC8,DC9;
    int A1, pred;

```

```

/* Force some input to be done if we are getting ahead of the input. */
while (cinfo->input_scan_number <= cinfo->output_scan_number &&
! cinfo->inputctl->eoi_reached) {
    if (cinfo->input_scan_number == cinfo->output_scan_number) {
        /* If input is working on current scan, we ordinarily want it to
         * have completed the current row. But if input scan is DC,
         * we want it to keep one row ahead so that next block row's DC
         * values are up to date.
         */
        JDIMENSION delta = (cinfo->Ss == 0) ? 1 : 0;
        if (cinfo->input_iMCU_row > cinfo->output_iMCU_row+delta)
            break;
    }
    if ((*cinfo->inputctl->consume_input)(cinfo) == JPEG_SUSPENDED)
        return JPEG_SUSPENDED;
}

/* OK, output from the virtual arrays. */
for (ci = 0, compptr = cinfo->comp_info; ci < cinfo->num_components;
    ci++, compptr++) {
    /* Don't bother to IDCT an uninteresting component. */
    if (! compptr->component_needed)
        continue;
    /* Count non-dummy DCT block rows in this iMCU row. */
    if (cinfo->output_iMCU_row < last_iMCU_row) {
        block_rows = compptr->v_samp_factor;
        access_rows = block_rows * 2; /* this and next iMCU row */
        last_row = FALSE;
    } else {
        /* NB: can't use last_row_height here; it is input-side-dependent! */
        block_rows = (int) (compptr->height_in_blocks % compptr->v_samp_factor);
        if (block_rows == 0) block_rows = compptr->v_samp_factor;
        access_rows = block_rows; /* this iMCU row only */
        last_row = TRUE;
    }
    /* Align the virtual buffer for this component. */
    if (cinfo->output_iMCU_row > 0) {
        access_rows += compptr->v_samp_factor; /* prior iMCU row too */
        buffer = (*cinfo->mem->access_virt_barray)
            ((j_common_ptr) cinfo, coef->whole_image[ci],
             (cinfo->output_iMCU_row - 1) * compptr->v_samp_factor,
             (JDIMENSION) access_rows, FALSE);
        buffer += compptr->v_samp_factor; /* point to current iMCU row */
        first_row = FALSE;
    } else {
        buffer = (*cinfo->mem->access_virt_barray)
            ((j_common_ptr) cinfo, coef->whole_image[ci],
             (JDIMENSION) 0, (JDIMENSION) access_rows, FALSE);
        first_row = TRUE;
    }
    /* Fetch component-dependent info */
    coef_bits = coef->coef_bits_latch + (ci * SAVED_COEFS);
    quanttbl = compptr->quant_table;
    Q00 = quanttbl->quantval[0];
    Q01 = quanttbl->quantval[Q01_POS];
    Q10 = quanttbl->quantval[Q10_POS];
    Q20 = quanttbl->quantval[Q20_POS];
    Q11 = quanttbl->quantval[Q11_POS];
    Q02 = quanttbl->quantval[Q02_POS];
    inverse_DCT = cinfo->idct->inverse_DCT[ci];
    output_ptr = output_buf[ci];
    /* Loop over all DCT blocks to be processed. */
    for (block_row = 0; block_row < block_rows; block_row++) {
        buffer_ptr = buffer[block_row];
        if (first_row && block_row == 0)
            prev_block_row = buffer_ptr;
        else
            prev_block_row = buffer[block_row-1];
        if (last_row && block_row == block_rows-1)
            next_block_row = buffer_ptr;
        else
            next_block_row = buffer[block_row+1];
        /* We fetch the surrounding DC values using a sliding-register approach.
         * Initialize all nine here so as to do the right thing on narrow pics.
         */
        DC1 = DC2 = DC3 = (int) prev_block_row[0][0];
        DC4 = DC5 = DC6 = (int) buffer_ptr[0][0];
        DC7 = DC8 = DC9 = (int) next_block_row[0][0];
        output_col = 0;
        last_block_column = compptr->width_in_blocks - 1;
    }
}

```

```

    for (block_num = 0; block_num <= last_block_column; block_num++) {
/* Fetch current DCT block into workspace so we can modify it */
jcopy_block_row(buffer_ptr, (JBLOCKROW) workspace, (JDIMENSION) 1);
/* Update DC values */
if (block_num < last_block_column) {
    DC3 = (int) prev_block_row[1][0];
    DC6 = (int) buffer_ptr[1][0];
    DC9 = (int) next_block_row[1][0];
}
/* Compute coefficient estimates per K.8.
 * An estimate is applied only if coefficient is still zero,
 * and is not known to be fully accurate.
 */
/* AC01 */
if ((Al=coef_bits[1]) != 0 && workspace[1] == 0) {
    num = 36 * Q00 * (DC4 - DC6);
    if (num >= 0) {
        pred = (int) (((Q01<<7) + num) / (Q01<<8));
        if (Al > 0 && pred >= (1<<Al))
            pred = (1<<Al)-1;
    } else {
        pred = (int) (((Q01<<7) - num) / (Q01<<8));
        if (Al > 0 && pred >= (1<<Al))
            pred = (1<<Al)-1;
        pred = -pred;
    }
    workspace[1] = (JCOEF) pred;
}
/* AC10 */
if ((Al=coef_bits[2]) != 0 && workspace[8] == 0) {
    num = 36 * Q00 * (DC2 - DC8);
    if (num >= 0) {
        pred = (int) (((Q10<<7) + num) / (Q10<<8));
        if (Al > 0 && pred >= (1<<Al))
            pred = (1<<Al)-1;
    } else {
        pred = (int) (((Q10<<7) - num) / (Q10<<8));
        if (Al > 0 && pred >= (1<<Al))
            pred = (1<<Al)-1;
        pred = -pred;
    }
    workspace[8] = (JCOEF) pred;
}
/* AC20 */
if ((Al=coef_bits[3]) != 0 && workspace[16] == 0) {
    num = 9 * Q00 * (DC2 + DC8 - 2*DC5);
    if (num >= 0) {
        pred = (int) (((Q20<<7) + num) / (Q20<<8));
        if (Al > 0 && pred >= (1<<Al))
            pred = (1<<Al)-1;
    } else {
        pred = (int) (((Q20<<7) - num) / (Q20<<8));
        if (Al > 0 && pred >= (1<<Al))
            pred = (1<<Al)-1;
        pred = -pred;
    }
    workspace[16] = (JCOEF) pred;
}
/* AC11 */
if ((Al=coef_bits[4]) != 0 && workspace[9] == 0) {
    num = 5 * Q00 * (DC1 - DC3 - DC7 + DC9);
    if (num >= 0) {
        pred = (int) (((Q11<<7) + num) / (Q11<<8));
        if (Al > 0 && pred >= (1<<Al))
            pred = (1<<Al)-1;
    } else {
        pred = (int) (((Q11<<7) - num) / (Q11<<8));
        if (Al > 0 && pred >= (1<<Al))
            pred = (1<<Al)-1;
        pred = -pred;
    }
    workspace[9] = (JCOEF) pred;
}
/* AC02 */
if ((Al=coef_bits[5]) != 0 && workspace[2] == 0) {
    num = 9 * Q00 * (DC4 + DC6 - 2*DC5);
    if (num >= 0) {
        pred = (int) (((Q02<<7) + num) / (Q02<<8));
        if (Al > 0 && pred >= (1<<Al))
            pred = (1<<Al)-1;
    }
}

```

```

    } else {
        pred = (int) (((Q02<<num) / (Q02<<8));
        if (A1 > 0 && pred >= 1<<A1))
            pred = (1<<A1)-1;
        pred = -pred;
    }
    workspace[2] = (JCOEF) pred;
}
/* OK, do the IDCT */
(*inverse_DCT) (cinfo, compptr, (JCOEFPTR) workspace,
                output_ptr, output_col);
/* Advance for next column */
DC1 = DC2; DC2 = DC3;
DC4 = DC5; DC5 = DC6;
DC7 = DC8; DC8 = DC9;
buffer_ptr++, prev_block_row++, next_block_row++;
output_col += compptr->DCT_scaled_size;
}
output_ptr += compptr->DCT_scaled_size;
}
}

if (++(cinfo->output_iMCU_row) < cinfo->total_iMCU_rows)
    return JPEG_ROW_COMPLETED;
return JPEG_SCAN_COMPLETED;
}

#endif /* BLOCK_SMOOTHING_SUPPORTED */

/*
 * Initialize coefficient buffer controller.
 */
GLOBAL(void)
jinit_d_coef_controller (j_decompress_ptr cinfo, boolean need_full_buffer)
{
    my_coef_ptr coef;

    coef = (my_coef_ptr)
        (*cinfo->mem->alloc_small) ((j_common_ptr) cinfo, JPOOL_IMAGE,
                                   SIZEOF(my_coef_controller));
    cinfo->coef = (struct jpeg_d_coef_controller *) coef;
    coef->pub.start_input_pass = start_input_pass;
    coef->pub.start_output_pass = start_output_pass;
#ifdef BLOCK_SMOOTHING_SUPPORTED
    coef->coef_bits_latch = NULL;
#endif

    /* Create the coefficient buffer. */
    if (need_full_buffer) {
#ifdef D_MULTISCAN_FILES_SUPPORTED
        /* Allocate a full-image virtual array for each component, */
        /* padded to a multiple of samp_factor DCT blocks in each direction. */
        /* Note we ask for a pre-zeroed array. */
        int ci, access_rows;
        jpeg_component_info *compptr;

        for (ci = 0, compptr = cinfo->comp_info; ci < cinfo->num_components;
             ci++, compptr++) {
            access_rows = compptr->v_samp_factor;
#ifdef BLOCK_SMOOTHING_SUPPORTED
            /* If block smoothing could be used, need a bigger window */
            if (cinfo->progressive_mode)
                access_rows *= 3;
#endif
            coef->whole_image[ci] = (*cinfo->mem->request_virt_barray)
                ((j_common_ptr) cinfo, JPOOL_IMAGE, TRUE,
                 (JDIMENSION) jround_up((long) compptr->width_in_blocks,
                                         (long) compptr->h_samp_factor),
                 (JDIMENSION) jround_up((long) compptr->height_in_blocks,
                                         (long) compptr->v_samp_factor),
                 (JDIMENSION) access_rows);
        }
        coef->pub.consume_data = consume_data;
        coef->pub.decompress_data = decompress_data;
        coef->pub.coef_arrays = coef->whole_image; /* link to virtual arrays */
    }
    #else
        ERREXIT(cinfo, JERR_NOT_COMPILED);
    #endif
}

```



```

    } else {
        /* We only need a single buffer. */
        JBLOCKROW buffer;
        int i;

        buffer = (JBLOCKROW)
            (*cinfo->mem->alloc_large) ((j_common_ptr) cinfo, JPOOL_IMAGE,
                D_MAX_BLOCKS_IN_MCU * sizeof(JBLOCK));
        for (i = 0; i < D_MAX_BLOCKS_IN_MCU; i++) {
            coef->MCU_buffer[i] = buffer + i;
        }
        coef->pub.consume_data = dummy_consume_data;
        coef->pub.decompress_data = decompress_onepass;
        coef->pub.coef_arrays = NULL; /* flag for no virtual arrays */
    }
}

```

[illegible]

```

/*
 * jdcolor.c
 *
 * Copyright (C) 1991-1997, Thomas G. Lane.
 * This file is part of the Independent JPEG Group's software.
 * For conditions of distribution and use, see the accompanying README file.
 *
 * This file contains output colorspace conversion routines.
 */

#define JPEG_INTERNALS
#include "jinclude.h"
#include "jpeglib.h"

/* Private subobject */

typedef struct {
  struct jpeg_color_deconverter pub; /* public fields */

  /* Private state for YCC->RGB conversion */
  int * Cr_r_tab; /* => table for Cr to R conversion */
  int * Cb_b_tab; /* => table for Cb to B conversion */
  INT32 * Cr_g_tab; /* => table for Cr to G conversion */
  INT32 * Cb_g_tab; /* => table for Cb to G conversion */
} my_color_deconverter;

typedef my_color_deconverter * my_cconvert_ptr;

/***** YCbCr -> RGB conversion: most common case *****/

/*
 * YCbCr is defined per CCIR 601-1, except that Cb and Cr are
 * normalized to the range 0..MAXJSAMPLE rather than -0.5 .. 0.5.
 * The conversion equations to be implemented are therefore
 *   R = Y + 1.40200 * Cr
 *   G = Y - 0.34414 * Cb - 0.71414 * Cr
 *   B = Y + 1.77200 * Cb
 * where Cb and Cr represent the incoming values less CENTERJSAMPLE.
 * (These numbers are derived from TIFF 6.0 section 21, dated 3-June-92.)
 *
 * To avoid floating-point arithmetic, we represent the fractional constants
 * as integers scaled up by 2^16 (about 4 digits precision); we have to divide
 * the products by 2^16, with appropriate rounding, to get the correct answer.
 * Notice that Y, being an integral input, does not contribute any fraction
 * so it need not participate in the rounding.
 *
 * For even more speed, we avoid doing any multiplications in the inner loop
 * by precalculating the constants times Cb and Cr for all possible values.
 * For 8-bit JSAMPLEs this is very reasonable (only 256 entries per table);
 * for 12-bit samples it is still acceptable. It's not very reasonable for
 * 16-bit samples, but if you want lossless storage you shouldn't be changing
 * colorspace anyway.
 * The Cr=>R and Cb=>B values can be rounded to integers in advance; the
 * values for the G calculation are left scaled up, since we must add them
 * together before rounding.
 */

#define SCALEBITS 16 /* speediest right-shift on some machines */
#define ONE_HALF ((INT32) 1 << (SCALEBITS-1))
#define FIX(x) ((INT32) ((x) * (1L<<SCALEBITS) + 0.5))

/*
 * Initialize tables for YCC->RGB colorspace conversion.
 */

LOCAL(void)
build_ycc_rgb_table (j_decompress_ptr cinfo)
{
  my_cconvert_ptr cconvert = (my_cconvert_ptr) cinfo->cconvert;
  int i;
  INT32 x;
  SHIFT_TEMPS

  cconvert->Cr_r_tab = (int *)
    (*cinfo->mem->alloc_small) ((j_common_ptr) cinfo, JPOOL_IMAGE,
                              (MAXJSAMPLE+1) * sizeof(int));
  cconvert->Cb_b_tab = (int *)

```

```

    (*cinfo->mem->alloc_small) ((j_common_ptr) cinfo, JPOOL_IMAGE,
    (MAXJSAMPLE+1) * sizeof(int));
cconvert->Cr_g_tab = (INT32 *)
    (*cinfo->mem->alloc_small) ((j_common_ptr) cinfo, JPOOL_IMAGE,
    (MAXJSAMPLE+1) * sizeof(INT32));
cconvert->Cb_g_tab = (INT32 *)
    (*cinfo->mem->alloc_small) ((j_common_ptr) cinfo, JPOOL_IMAGE,
    (MAXJSAMPLE+1) * sizeof(INT32));

for (i = 0, x = -CENTERJSAMPLE; i <= MAXJSAMPLE; i++, x++) {
    /* i is the actual input pixel value, in the range 0..MAXJSAMPLE */
    /* The Cb or Cr value we are thinking of is x = i - CENTERJSAMPLE */
    /* Cr=>R value is nearest int to 1.40200 * x */
    cconvert->Cr_r_tab[i] = (int)
        RIGHT_SHIFT(FIX(1.40200) * x + ONE_HALF, SCALEBITS);
    /* Cb=>B value is nearest int to 1.77200 * x */
    cconvert->Cb_b_tab[i] = (int)
        RIGHT_SHIFT(FIX(1.77200) * x + ONE_HALF, SCALEBITS);
    /* Cr=>G value is scaled-up -0.71414 * x */
    cconvert->Cr_g_tab[i] = (- FIX(0.71414)) * x;
    /* Cb=>G value is scaled-up -0.34414 * x */
    /* We also add in ONE_HALF so that need not do it in inner loop */
    cconvert->Cb_g_tab[i] = (- FIX(0.34414)) * x + ONE_HALF;
}

/*
 * Convert some rows of samples to the output colorspace.
 *
 * Note that we change from noninterleaved, one-plane-per-component format
 * to interleaved-pixel format. The output buffer is therefore three times
 * as wide as the input buffer.
 * A starting row offset is provided only for the input buffer. The caller
 * can easily adjust the passed output_buf value to accommodate any row
 * offset required on that side.
 */
METHODDEF(void)
ycc_rgb_convert (j_decompress_ptr cinfo,
    JSAMPIMAGE input_buf, JDIMENSION input_row,
    JSAMPARRAY output_buf, int num_rows)
{
    my_cconvert_ptr cconvert = (my_cconvert_ptr) cinfo->cconvert;
    register int y, cb, cr;
    register JSAMPROW outptr;
    register JSAMPROW inptr0, inptr1, inptr2;
    register JDIMENSION col;
    JDIMENSION num_cols = cinfo->output_width;
    /* copy these pointers into registers if possible */
    register JSAMPLE * range_limit = cinfo->sample_range_limit;
    register int * Crrtab = cconvert->Cr_r_tab;
    register int * Cbbtab = cconvert->Cb_b_tab;
    register INT32 * Crgtab = cconvert->Cr_g_tab;
    register INT32 * Cbgtab = cconvert->Cb_g_tab;
    SHIFT_TEMPS

    while (--num_rows >= 0) {
        inptr0 = input_buf[0][input_row];
        inptr1 = input_buf[1][input_row];
        inptr2 = input_buf[2][input_row];
        input_row++;
        outptr = *output_buf++;
        for (col = 0; col < num_cols; col++) {
            y = GETJSAMPLE(inptr0[col]);
            cb = GETJSAMPLE(inptr1[col]);
            cr = GETJSAMPLE(inptr2[col]);
            /* Range-limiting is essential due to noise introduced by DCT losses. */
            outptr[RGB_RED] = range_limit[y + Crrtab[cr]];
            outptr[RGB_GREEN] = range_limit[y +
                ((int) RIGHT_SHIFT(Cbgtab[cb] + Crgtab[cr],
                    SCALEBITS))];
            outptr[RGB_BLUE] = range_limit[y + Cbbtab[cb]];
            outptr += RGB_PIXELSIZE;
        }
    }
}

/***** Cases other than YCbCr -> RGB *****/

```

```

/*
 * Color conversion for no colorspace change: just copy the data,
 * converting from separate-planes to interleaved representation.
 */

```

```

METHODDEF(void)
null_convert (j_decompress_ptr cinfo,
              JSAMPIMAGE input_buf, JDIMENSION input_row,
              JSAMPARRAY output_buf, int num_rows)
{
    register JSAMPROW inptr, outptr;
    register JDIMENSION count;
    register int num_components = cinfo->num_components;
    JDIMENSION num_cols = cinfo->output_width;
    int ci;

    while (--num_rows >= 0) {
        for (ci = 0; ci < num_components; ci++) {
            inptr = input_buf[ci][input_row];
            outptr = output_buf[0] + ci;
            for (count = num_cols; count > 0; count--) {
                *outptr = *inptr++; /* needn't bother with GETJSAMPLE() here */
                outptr += num_components;
            }
            input_row++;
            output_buf++;
        }
    }
}

```

```

/*
 * Color conversion for grayscale: just copy the data.
 * This also works for YCbCr -> grayscale conversion, in which
 * we just copy the Y (luminance) component and ignore chrominance.
 */

```

```

METHODDEF(void)
grayscale_convert (j_decompress_ptr cinfo,
                  JSAMPIMAGE input_buf, JDIMENSION input_row,
                  JSAMPARRAY output_buf, int num_rows)
{
    jcopy_sample_rows(input_buf[0], (int) input_row, output_buf, 0,
                      num_rows, cinfo->output_width);
}

```

```

/*
 * Convert grayscale to RGB: just duplicate the graylevel three times.
 * This is provided to support applications that don't want to cope
 * with grayscale as a separate case.
 */

```

```

METHODDEF(void)
gray_rgb_convert (j_decompress_ptr cinfo,
                  JSAMPIMAGE input_buf, JDIMENSION input_row,
                  JSAMPARRAY output_buf, int num_rows)
{
    register JSAMPROW inptr, outptr;
    register JDIMENSION col;
    JDIMENSION num_cols = cinfo->output_width;

    while (--num_rows >= 0) {
        inptr = input_buf[0][input_row++];
        outptr = *output_buf++;
        for (col = 0; col < num_cols; col++) {
            /* We can dispense with GETJSAMPLE() here */
            outptr[RGB_RED] = outptr[RGB_GREEN] = outptr[RGB_BLUE] = inptr[col];
            outptr += RGB_PIXELSIZE;
        }
    }
}

```

```

/*
 * Adobe-style YCKK->CMYK conversion.
 * We convert YCbCr to R=1-C, G=1-M, and B=1-Y using the same
 * conversion as above, while passing K (black) unchanged.
 */

```

```

* We assume build_ycc_rgb_tab has been called.
*/

```

```

METHODDEF(void)

```

```

yccck_cmyk_convert (j_decompress_ptr cinfo,
                    JSAMPIMAGE input_buf, JDIMENSION input_row,
                    JSAMPARRAY output_buf, int num_rows)

```

```

{
    my_cconvert_ptr cconvert = (my_cconvert_ptr) cinfo->cconvert;
    register int y, cb, cr;
    register JSAMPROW outptr;
    register JSAMPROW inptr0, inptr1, inptr2, inptr3;
    register JDIMENSION col;
    JDIMENSION num_cols = cinfo->output_width;
    /* copy these pointers into registers if possible */
    register JSAMPLE * range_limit = cinfo->sample_range_limit;
    register int * Crrtab = cconvert->Cr_r_tab;
    register int * Cbbtab = cconvert->Cb_b_tab;
    register INT32 * Crgtab = cconvert->Cr_g_tab;
    register INT32 * Cbgtab = cconvert->Cb_g_tab;
    SHIFT_TEMPS

    while (--num_rows >= 0) {
        inptr0 = input_buf[0][input_row];
        inptr1 = input_buf[1][input_row];
        inptr2 = input_buf[2][input_row];
        inptr3 = input_buf[3][input_row];
        input_row++;
        outptr = *output_buf++;
        for (col = 0; col < num_cols; col++) {
            y = GETJSAMPLE(inptr0[col]);
            cb = GETJSAMPLE(inptr1[col]);
            cr = GETJSAMPLE(inptr2[col]);
            /* Range-limiting is essential due to noise introduced by DCT losses. */
            outptr[0] = range_limit[MAXJSAMPLE - (y + Crrtab[cr])]; /* red */
            outptr[1] = range_limit[MAXJSAMPLE - (y + Cbgtab[cb] + Crgtab[cr], /* green */
                                           ((int) RIGHT_SHIFT(Cbgtab[cb] + Crgtab[cr],
                                           SCALEBITS))]);
            outptr[2] = range_limit[MAXJSAMPLE - (y + Cbbtab[cb])]; /* blue */
            /* K passes through unchanged */
            outptr[3] = inptr3[col]; /* don't need GETJSAMPLE here */
            outptr += 4;
        }
    }
}

```

```

/* Empty method for start_pass.
*/

```

```

METHODDEF(void)

```

```

start_pass_dcolor (j_decompress_ptr cinfo)

```

```

{
    /* no work needed */
}

```

```

/*
* Module initialization routine for output colorspace conversion.
*/

```

```

GLOBAL(void)

```

```

jinit_color_deconverter (j_decompress_ptr cinfo)

```

```

{
    my_cconvert_ptr cconvert;
    int ci;

    cconvert = (my_cconvert_ptr)
        (*cinfo->mem->alloc_small) ((j_common_ptr) cinfo, JPOOL_IMAGE,
        SIZEOF(my_color_deconverter));
    cinfo->cconvert = (struct jpeg_color_deconverter *) cconvert;
    cconvert->pub.start_pass = start_pass_dcolor;

    /* Make sure num_components agrees with jpeg_color_space */
    switch (cinfo->jpeg_color_space) {
    case JCS_GRAYSCALE:
        if (cinfo->num_components != 1)
            ERREXIT(cinfo, JERR_BAD_J_COLORSPACE);
        break;

```

```

case JCS_RGB:
case JCS_YCbCr:
    if (cinfo->num_components != 3)
        ERREXIT(cinfo, JERR_BAD_J_COLORSPACE);
    break;

case JCS_CMYK:
case JCS_YCCK:
    if (cinfo->num_components != 4)
        ERREXIT(cinfo, JERR_BAD_J_COLORSPACE);
    break;

default:
    /* JCS_UNKNOWN can be anything */
    if (cinfo->num_components < 1)
        ERREXIT(cinfo, JERR_BAD_J_COLORSPACE);
    break;
}

/* Set out_color_components and conversion method based on requested space.
 * Also clear the component_needed flags for any unused components,
 * so that earlier pipeline stages can avoid useless computation.
 */

switch (cinfo->out_color_space) {
case JCS_GRAYSCALE:
    cinfo->out_color_components = 1;
    if (cinfo->jpeg_color_space == JCS_GRAYSCALE ||
        cinfo->jpeg_color_space == JCS_YCbCr) {
        cconvert->pub.color_convert = grayscale_convert;
        /* For color->grayscale conversion, only the Y (0) component is needed */
        for (ci = 1; ci < cinfo->num_components; ci++)
            cinfo->comp_info[ci].component_needed = FALSE;
    } else
        ERREXIT(cinfo, JERR_CONVERSION_NOTIMPL);
    break;

case JCS_RGB:
    cinfo->out_color_components = RGB_PIXELSIZE;
    if (cinfo->jpeg_color_space == JCS_YCbCr) {
        cconvert->pub.color_convert = ycc_rgb_convert;
        build_ycc_rgb_table(cinfo);
    } else if (cinfo->jpeg_color_space == JCS_GRAYSCALE) {
        cconvert->pub.color_convert = gray_rgb_convert;
    } else if (cinfo->jpeg_color_space == JCS_RGB && RGB_PIXELSIZE == 3) {
        cconvert->pub.color_convert = null_convert;
    } else
        ERREXIT(cinfo, JERR_CONVERSION_NOTIMPL);
    break;

case JCS_CMYK:
    cinfo->out_color_components = 4;
    if (cinfo->jpeg_color_space == JCS_YCCK) {
        cconvert->pub.color_convert = ycck_cmyk_convert;
        build_ycc_rgb_table(cinfo);
    } else if (cinfo->jpeg_color_space == JCS_CMYK) {
        cconvert->pub.color_convert = null_convert;
    } else
        ERREXIT(cinfo, JERR_CONVERSION_NOTIMPL);
    break;

default:
    /* Permit null conversion to same output space */
    if (cinfo->out_color_space == cinfo->jpeg_color_space) {
        cinfo->out_color_components = cinfo->num_components;
        cconvert->pub.color_convert = null_convert;
    } else
        /* unsupported non-null conversion */
        ERREXIT(cinfo, JERR_CONVERSION_NOTIMPL);
    break;
}

if (cinfo->quantize_colors)
    cinfo->output_components = 1; /* single colormapped output component */
else
    cinfo->output_components = cinfo->out_color_components;
}

```

```

/*
 * jddctmgr.c
 *
 * Copyright (C) 1994-1996, Thomas G. Lane.
 * This file is part of the Independent JPEG Group's software.
 * For conditions of distribution and use, see the accompanying README file.
 *
 * This file contains the inverse-DCT management logic.
 * This code selects a particular IDCT implementation to be used,
 * and it performs related housekeeping chores.  No code in this file
 * is executed per IDCT step, only during output pass setup.
 *
 * Note that the IDCT routines are responsible for performing coefficient
 * dequantization as well as the IDCT proper.  This module sets up the
 * dequantization multiplier table needed by the IDCT routine.
 */

#define JPEG_INTERNALS
#include "jinclude.h"
#include "jpeglib.h"
#include "jdct.h"          /* Private declarations for DCT subsystem */

/*
 * The decompressor input side (jinput.c) saves away the appropriate
 * quantization table for each component at the start of the first scan
 * involving that component.  (This is necessary in order to correctly
 * decode files that reuse Q-table slots.)
 * When we are ready to make an output pass, the saved Q-table is converted
 * to a multiplier table that will actually be used by the IDCT routine.
 * The multiplier table contents are IDCT-method-dependent.  To support
 * application changes in IDCT method between scans, we can remake the
 * multiplier tables if necessary.
 * In buffered-image mode, the first output pass may occur before any data
 * has been seen for some components, and thus before their Q-tables have
 * been saved away.  To handle this case, multiplier tables are preset
 * to zeroes; the result of the IDCT will be a neutral gray level.
 */

/* Private subobject for this module */
typedef struct {
  struct jpeg_inverse_dct pub; /* public fields */
  /* This array contains the IDCT method code that each multiplier table
   * is currently set up for, or -1 if it's not yet set up.
   * The actual multiplier tables are pointed to by dct_table in the
   * per-component comp_info structures.
   */
  int cur_method[MAX_COMPONENTS];
  my_idct_controller;
} my_idct_controller;

typedef my_idct_controller * my_idct_ptr;

/* Allocated multiplier tables: big enough for any supported variant */

typedef union {
  ISLOW_MULT_TYPE islow_array[DCTSIZE2];
#ifdef DCT_IFAST_SUPPORTED
  IFAST_MULT_TYPE ifast_array[DCTSIZE2];
#endif
#ifdef DCT_FLOAT_SUPPORTED
  FLOAT_MULT_TYPE float_array[DCTSIZE2];
#endif
} multiplier_table;

/* The current scaled-IDCT routines require ISLOW-style multiplier tables,
 * so be sure to compile that code if either ISLOW or SCALING is requested.
 */
#ifdef DCT_ISLOW_SUPPORTED
#define PROVIDE_ISLOW_TABLES
#else
#ifdef IDCT_SCALING_SUPPORTED
#define PROVIDE_ISLOW_TABLES
#endif
#endif

```

```

/*
 * Prepare for an output pass.
 * Here we select the proper IDCT routine for each component and build
 * a matching multiplier table.
 */

```

```

METHODDEF(void)
start_pass (j_decompress_ptr cinfo)
{
    my_idct_ptr idct = (my_idct_ptr) cinfo->idct;
    int ci, i;
    jpeg_component_info *comp_ptr;
    int method = 0;
    inverse_DCT_method_ptr method_ptr = NULL;
    JQUANT_TBL * qtbl;

    for (ci = 0, comp_ptr = cinfo->comp_info; ci < cinfo->num_components;
        ci++, comp_ptr++) {
        /* Select the proper IDCT routine for this component's scaling */
        switch (comp_ptr->DCT_scaled_size) {
#ifdef IDCT_SCALING_SUPPORTED
        case 1:
            method_ptr = jpeg_idct_1x1;
            method = JDCT_ISLOW; /* jidctred uses islow-style table */
            break;
        case 2:
            method_ptr = jpeg_idct_2x2;
            method = JDCT_ISLOW; /* jidctred uses islow-style table */
            break;
        case 4:
            method_ptr = jpeg_idct_4x4;
            method = JDCT_ISLOW; /* jidctred uses islow-style table */
            break;
#endif
        case DCTSIZE:
            switch (cinfo->dct_method) {
#ifdef DCT_ISLOW_SUPPORTED
            case JDCT_ISLOW:
                method_ptr = jpeg_idct_islow;
                method = JDCT_ISLOW;
                break;
#endif
#ifdef DCT_IFAST_SUPPORTED
            case JDCT_IFAST:
                method_ptr = jpeg_idct_ifast;
                method = JDCT_IFAST;
                break;
#endif
#ifdef DCT_FLOAT_SUPPORTED
            case JDCT_FLOAT:
                method_ptr = jpeg_idct_float;
                method = JDCT_FLOAT;
                break;
#endif
            default:
                ERREXIT(cinfo, JERR_NOT_COMPILED);
                break;
            }
        default:
            ERREXIT1(cinfo, JERR_BAD_DCTSIZE, comp_ptr->DCT_scaled_size);
            break;
        }
        idct->pub.inverse_DCT[ci] = method_ptr;
        /* Create multiplier table from quant table.
         * However, we can skip this if the component is uninteresting
         * or if we already built the table. Also, if no quant table
         * has yet been saved for the component, we leave the
         * multiplier table all-zero; we'll be reading zeroes from the
         * coefficient controller's buffer anyway.
         */
        if (!comp_ptr->component_needed || idct->cur_method[ci] == method)
            continue;
        qtbl = comp_ptr->quant_table;
        if (qtbl == NULL) /* happens if no data yet for component */
            continue;
        idct->cur_method[ci] = method;
        switch (method) {
#ifdef PROVIDE_ISLOW_TABLES

```



```

case JDCT_ISLOW:
{
/* For LL&M IDCT method, multipliers are equal to raw quantization
 * coefficients, but are stored as ints to ensure access efficiency.
 */
ISLOW_MULT_TYPE * ismtbl = (ISLOW_MULT_TYPE *) compptr->dct_table;
for (i = 0; i < DCTSIZE2; i++) {
    ismtbl[i] = (ISLOW_MULT_TYPE) qtbl->quantval[i];
}
}
break;
#endif
#ifdef DCT_IFAST_SUPPORTED
case JDCT_IFAST:
{
/* For AA&N IDCT method, multipliers are equal to quantization
 * coefficients scaled by scalefactor[row]*scalefactor[col], where
 * scalefactor[0] = 1
 * scalefactor[k] = cos(k*PI/16) * sqrt(2) for k=1..7
 * For integer operation, the multiplier table is to be scaled by
 * IFAST_SCALE_BITS.
 */
IFAST_MULT_TYPE * ifmtbl = (IFAST_MULT_TYPE *) compptr->dct_table;
#define CONST_BITS 14
static const INT16 aanscales[DCTSIZE2] = {
/* precomputed values scaled up by 14 bits */
16384, 22725, 21407, 19266, 16384, 12873, 8867, 4520,
22725, 31521, 29692, 26722, 22725, 17855, 12299, 6270,
21407, 29692, 27969, 25172, 21407, 16819, 11585, 5906,
19266, 26722, 25172, 22654, 19266, 15137, 10426, 5315,
16384, 22725, 21407, 19266, 16384, 12873, 8867, 4520,
12873, 17855, 16819, 15137, 12873, 10114, 6967, 3552,
8867, 12299, 11585, 10426, 8867, 6967, 4799, 2446,
4520, 6270, 5906, 5315, 4520, 3552, 2446, 1247
};
SHIFT_TEMPS
for (i = 0; i < DCTSIZE2; i++) {
    ifmtbl[i] = (IFAST_MULT_TYPE)
        DESCALE(MULTIPLY16V16((INT32) qtbl->quantval[i],
            (INT32) aanscales[i]),
            CONST_BITS-IFAST_SCALE_BITS);
}
}
break;
#endif
#ifdef DCT_FLOAT_SUPPORTED
case JDCT_FLOAT:
{
/* For float AA&N IDCT method, multipliers are equal to quantization
 * coefficients scaled by scalefactor[row]*scalefactor[col], where
 * scalefactor[0] = 1
 * scalefactor[k] = cos(k*PI/16) * sqrt(2) for k=1..7
 */
FLOAT_MULT_TYPE * fmbtbl = (FLOAT_MULT_TYPE *) compptr->dct_table;
int row, col;
static const double aanscalefactor[DCTSIZE] = {
1.0, 1.387039845, 1.306562965, 1.175875602,
1.0, 0.785694958, 0.541196100, 0.275899379
};
i = 0;
for (row = 0; row < DCTSIZE; row++) {
    for (col = 0; col < DCTSIZE; col++) {
        fmbtbl[i] = (FLOAT_MULT_TYPE)
            ((double) qtbl->quantval[i] *
            aanscalefactor[row] * aanscalefactor[col]);
        i++;
    }
}
break;
#endif
default:
    ERREXIT(cinfo, JERR_NOT_COMPILED);
    break;
}
}

```

```

/*
 * Initialize IDCT manager.
 */

GLOBAL(void)
jinit_inverse_dct (j_decompress_ptr cinfo)
{
    my_idct_ptr idct;
    int ci;
    jpeg_component_info *comp_ptr;

    idct = (my_idct_ptr)
        (*cinfo->mem->alloc_small) ((j_common_ptr) cinfo, JPOOL_IMAGE,
            SIZEOF(my_idct_controller));
    cinfo->idct = (struct jpeg_inverse_dct *) idct;
    idct->pub.start_pass = start_pass;

    for (ci = 0, comp_ptr = cinfo->comp_info; ci < cinfo->num_components;
        ci++, comp_ptr++) {
        /* Allocate and pre-zero a multiplier table for each component */
        comp_ptr->dct_table =
            (*cinfo->mem->alloc_small) ((j_common_ptr) cinfo, JPOOL_IMAGE,
                SIZEOF(multiplier_table));
        MEMZERO(comp_ptr->dct_table, SIZEOF(multiplier_table));
        /* Mark multiplier table not yet set up for any method */
        idct->cur_method[ci] = -1;
    }
}

```

```

/*
 * jdhuuff.c
 *
 * Copyright (C) 1991-1997, Thomas G. Lane.
 * This file is part of the Independent JPEG Group's software.
 * For conditions of distribution and use, see the accompanying README file.
 *
 * This file contains Huffman entropy decoding routines.
 *
 * Much of the complexity here has to do with supporting input suspension.
 * If the data source module demands suspension, we want to be able to back
 * up to the start of the current MCU. To do this, we copy state variables
 * into local working storage, and update them back to the permanent
 * storage only upon successful completion of an MCU.
 */

#define JPEG_INTERNALS
#include "jinclude.h"
#include "jpeglib.h"
#include "jdhuuff.h" /* Declarations shared with jdphuff.c */

/*
 * Expanded entropy decoder object for Huffman decoding.
 *
 * The savable_state subrecord contains fields that change within an MCU,
 * but must not be updated permanently until we complete the MCU.
 */

typedef struct {
  int last_dc_val[MAX_COMPS_IN_SCAN]; /* last DC coef for each component */
} savable_state;

/* This macro is to work around compilers with missing or broken
 * structure assignment. You'll need to fix this code if you have
 * such a compiler and you change MAX_COMPS_IN_SCAN. */
#ifdef NO_STRUCT_ASSIGN
#define NO_STRUCT_ASSIGN
#endif
#define NO_STRUCT_ASSIGN
#define ASSIGN_STATE(dest,src) ((dest) = (src))
#else
#define ASSIGN_STATE(dest,src) \
  if (MAX_COMPS_IN_SCAN == 4) \
  { \
    (dest).last_dc_val[0] = (src).last_dc_val[0], \
    (dest).last_dc_val[1] = (src).last_dc_val[1], \
    (dest).last_dc_val[2] = (src).last_dc_val[2], \
    (dest).last_dc_val[3] = (src).last_dc_val[3]; \
  }
#endif

typedef struct {
  struct jpeg_entropy_decoder pub; /* public fields */

  /* These fields are loaded into local variables at start of each MCU.
   * In case of suspension, we exit WITHOUT updating them.
   */
  bitread_perm_state bitstate; /* Bit buffer at start of MCU */
  savable_state saved; /* Other state at start of MCU */

  /* These fields are NOT loaded into local working state. */
  unsigned int restarts_to_go; /* MCUs left in this restart interval */

  /* Pointers to derived tables (these workspaces have image lifespan) */
  d_derived_tbl * dc_derived_tbls[NUM_HUFF_TBLS];
  d_derived_tbl * ac_derived_tbls[NUM_HUFF_TBLS];

  /* Precalculated info set up by start_pass for use in decode_mcu: */

  /* Pointers to derived tables to be used for each block within an MCU */
  d_derived_tbl * dc_cur_tbls[D_MAX_BLOCKS_IN_MCU];
  d_derived_tbl * ac_cur_tbls[D_MAX_BLOCKS_IN_MCU];
  /* Whether we care about the DC and AC coefficient values for each block */
  boolean dc_needed[D_MAX_BLOCKS_IN_MCU];
  boolean ac_needed[D_MAX_BLOCKS_IN_MCU];
} huff_entropy_decoder;

typedef huff_entropy_decoder * huff_entropy_ptr;

```

```

/*
 * Initialize for a Huffman-coded scan.
 */

METHODDEF(void)
start_pass_huff_decoder (j_decompress_ptr cinfo)
{
    huff_entropy_ptr entropy = (huff_entropy_ptr) cinfo->entropy;
    int ci, blkcn, dctbl, actbl;
    jpeg_component_info * compptr;

    /* Check that the scan parameters Ss, Se, Ah/Al are OK for sequential JPEG.
     * This ought to be an error condition, but we make it a warning because
     * there are some baseline files out there with all zeroes in these bytes.
     */
    if (cinfo->Ss != 0 || cinfo->Se != DCTSIZE2-1 ||
        cinfo->Ah != 0 || cinfo->Al != 0)
        WARNMS(cinfo, JWRN_NOT_SEQUENTIAL);

    for (ci = 0; ci < cinfo->comps_in_scan; ci++) {
        compptr = cinfo->cur_comp_info[ci];
        dctbl = compptr->dc_tbl_no;
        actbl = compptr->ac_tbl_no;
        /* Compute derived values for Huffman tables */
        /* We may do this more than once for a table, but it's not expensive */
        jpeg_make_d_derived_tbl(cinfo, TRUE, dctbl,
                                & entropy->dc_derived_tbls[dctbl]);
        jpeg_make_d_derived_tbl(cinfo, FALSE, actbl,
                                & entropy->ac_derived_tbls[actbl]);
        /* Initialize DC predictions to 0 */
        entropy->saved.last_dc_val[ci] = 0;

        /* Precalculate decoding info for each block in an MCU of this scan */
        for (blkcn = 0; blkcn < cinfo->blocks_in_MCU; blkcn++) {
            ci = cinfo->MCU_membership[blkcn];
            compptr = cinfo->cur_comp_info[ci];
            /* Precalculate which table to use for each block */
            entropy->dc_cur_tbls[blkcn] = entropy->dc_derived_tbls[compptr->dc_tbl_no];
            entropy->ac_cur_tbls[blkcn] = entropy->ac_derived_tbls[compptr->ac_tbl_no];
            /* Decide whether we really care about the coefficient values */
            if (compptr->component_needed) {
                entropy->dc_needed[blkcn] = TRUE;
                /* we don't need the ACs if producing a 1/8th-size image */
                entropy->ac_needed[blkcn] = (compptr->DCT_scaled_size > 1);
            } else {
                entropy->dc_needed[blkcn] = entropy->ac_needed[blkcn] = FALSE;
            }
        }

        /* Initialize bitread state variables */
        entropy->bitstate.bits_left = 0;
        entropy->bitstate.get_buffer = 0; /* unnecessary, but keeps Purify quiet */
        entropy->pub.insufficient_data = FALSE;

        /* Initialize restart counter */
        entropy->restarts_to_go = cinfo->restart_interval;
    }

    /*
     * Compute the derived values for a Huffman table.
     * This routine also performs some validation checks on the table.
     *
     * Note this is also used by jdphuff.c.
     */
}

GLOBAL(void)
jpeg_make_d_derived_tbl (j_decompress_ptr cinfo, boolean isDC, int tblno,
                        d_derived_tbl ** pdtbl)
{
    JHUFF_TBL *htbl;
    d_derived_tbl *dtbl;
    int p, i, l, si, numsymbols;
    int lookbits, ctr;
    char huffsize[257];
    unsigned int huffcode[257];
    unsigned int code;

    /* Note that huffsize[] and huffcode[] are filled in code-length order,

```

```

/* paralleling the order of the symbols themselves in htbl->huff[] */
/*
/* Find the input Huffman table */
if (tblno < 0 || tblno >= NUM_HUFF_TBLS)
    ERREXIT1(cinfo, JERR_NO_HUFF_TABLE, tblno);
htbl =
    isDC ? cinfo->dc_huff_tbl_ptrs[tblno] : cinfo->ac_huff_tbl_ptrs[tblno];
if (htbl == NULL)
    ERREXIT1(cinfo, JERR_NO_HUFF_TABLE, tblno);

/* Allocate a workspace if we haven't already done so. */
if (*pdtbl == NULL)
    *pdtbl = (d_derived_tbl *)
        (*cinfo->mem->alloc_small) ((j_common_ptr) cinfo, JPOOL_IMAGE,
            sizeof(d_derived_tbl));
dtbl = *pdtbl;
dtbl->pub = htbl;      /* fill in back link */

/* Figure C.1: make table of Huffman code length for each symbol */

p = 0;
for (l = 1; l <= 16; l++) {
    i = (int) htbl->bits[l];
    if (i < 0 || p + i > 256) /* protect against table overrun */
        ERREXIT(cinfo, JERR_BAD_HUFF_TABLE);
    while (i--)
        huffsize[p++] = (char) l;
}
huffsize[p] = 0;
numsymbols = p;

/* Figure C.2: generate the codes themselves */
/* We also validate that the counts represent a legal Huffman code tree. */
code = 0;
si = huffsize[0];
p = 0;
while (huffsize[p]) {
    while (((int) huffsize[p]) == si) {
        huffcode[p++] = code;
        code++;
    }
    /* code is now 1 more than the last code used for codelength si; but
     * it must still fit in si bits, since no code is allowed to be all ones.
     */
    if (((INT32) code) >= (((INT32) 1) << si))
        ERREXIT(cinfo, JERR_BAD_HUFF_TABLE);
    code <<= 1;
    si++;
}

/* Figure F.15: generate decoding tables for bit-sequential decoding */

p = 0;
for (l = 1; l <= 16; l++) {
    if (htbl->bits[l]) {
        /* valoffset[l] = huffval[] index of 1st symbol of code length l,
         * minus the minimum code of length l
         */
        dtbl->valoffset[l] = (INT32) p - (INT32) huffcode[p];
        p += htbl->bits[l];
        dtbl->maxcode[l] = huffcode[p-1]; /* maximum code of length l */
    } else {
        dtbl->maxcode[l] = -1; /* -1 if no codes of this length */
    }
}
dtbl->maxcode[17] = 0xFFFFFL; /* ensures jpeg_huff_decode terminates */

/* Compute lookahead tables to speed up decoding.
 * First we set all the table entries to 0, indicating "too long";
 * then we iterate through the Huffman codes that are short enough and
 * fill in all the entries that correspond to bit sequences starting
 * with that code.
 */
MEMZERO(dtbl->look_nbits, sizeof(dtbl->look_nbits));

p = 0;
for (l = 1; l <= HUFF_LOOKAHEAD; l++) {

```

```

for (i = 1; i <= (int) htbl->bits[1]; i++, p++) {
    /* l = current code's length, p = its index in huffcode[] & huffval[]. */
    /* Generate left-justified code followed by all possible bit sequences */
    lookbits = huffcode[p] << (HUFF_LOOKAHEAD-1);
    for (ctr = 1 << (HUFF_LOOKAHEAD-1); ctr > 0; ctr--) {
        dtbl->look_nbits[lookbits] = 1;
        dtbl->look_sym[lookbits] = htbl->huffval[p];
        lookbits++;
    }
}

/* Validate symbols as being reasonable.
 * For AC tables, we make no check, but accept all byte values 0..255.
 * For DC tables, we require the symbols to be in range 0..15.
 * (Tighter bounds could be applied depending on the data depth and mode,
 * but this is sufficient to ensure safe decoding.)
 */
if (isDC) {
    for (i = 0; i < numsymbols; i++) {
        int sym = htbl->huffval[i];
        if (sym < 0 || sym > 15)
            ERREXIT(cinfo, JERR_BAD_HUFF_TABLE);
    }
}

/*
 * Out-of-line code for bit fetching (shared with jdphuff.c).
 * See jdphuff.h for info about usage.
 * Note: current values of get_buffer and bits_left are passed as parameters,
 * but are returned in the corresponding fields of the state struct.
 *
 * On most machines MIN_GET_BITS should be 25 to allow the full 32-bit width
 * of get_buffer to be used. (On machines with wider words, an even larger
 * buffer could be used.) However, on some machines 32-bit shifts are
 * quite slow and take time proportional to the number of places shifted.
 * (This is true with most PC compilers, for instance.) In this case it may
 * be a win to set MIN_GET_BITS to the minimum value of 15. This reduces the
 * average shift distance at the cost of more calls to jpeg_fill_bit_buffer.
 */
#ifdef SLOW_SHIFT_32
#define MIN_GET_BITS 15 /* minimum allowable value */
#else
#define MIN_GET_BITS (BIT_BUF_SIZE-7)
#endif

GLOBAL(boolean)
jpeg_fill_bit_buffer (bitread_working_state * state,
                    register bit_buf_type get_buffer, register int bits_left,
                    int nbits)
/* Load up the bit buffer to a depth of at least nbits */
{
    /* Copy heavily used state fields into locals (hopefully registers) */
    register const JOCTET * next_input_byte = state->next_input_byte;
    register size_t bytes_in_buffer = state->bytes_in_buffer;
    jpeg_decompress_ptr cinfo = state->cinfo;

    /* Attempt to load at least MIN_GET_BITS bits into get_buffer. */
    /* (It is assumed that no request will be for more than that many bits.) */
    /* We fail to do so only if we hit a marker or are forced to suspend. */

    if (cinfo->unread_marker == 0) { /* cannot advance past a marker */
        while (bits_left < MIN_GET_BITS) {
            register int c;

            /* Attempt to read a byte */
            if (bytes_in_buffer == 0) {
                if (! (*cinfo->src->fill_input_buffer) (cinfo))
                    return FALSE;
                next_input_byte = cinfo->src->next_input_byte;
                bytes_in_buffer = cinfo->src->bytes_in_buffer;
            }
            bytes_in_buffer--;
            c = GETJOCTET(*next_input_byte++);

            /* If it's 0xFF, check and discard stuffed zero byte */

```

```

    if (c == 0xFF) {
/* Loop here to discard a FF's on terminating marker
 * so that we can save a valid unread_marker value. NOTE: we will
 * accept multiple FF's followed by a 0 as meaning a single FF data
 * byte. This data pattern is not valid according to the standard.
 */
do {
    if (bytes_in_buffer == 0) {
        if (! (*cinfo->src->fill_input_buffer) (cinfo))
            return FALSE;
        next_input_byte = cinfo->src->next_input_byte;
        bytes_in_buffer = cinfo->src->bytes_in_buffer;
    }
    bytes_in_buffer--;
    c = GETJOCTET(*next_input_byte++);
} while (c == 0xFF);

if (c == 0) {
/* Found FF/00, which represents an FF data byte */
    c = 0xFF;
} else {
/* Oops, it's actually a marker indicating end of compressed data.
 * Save the marker code for later use.
 * Fine point: it might appear that we should save the marker into
 * bitread working state, not straight into permanent state. But
 * once we have hit a marker, we cannot need to suspend within the
 * current MCU, because we will read no more bytes from the data
 * source. So it is OK to update permanent state right away.
 */
    cinfo->unread_marker = c;
/* See if we need to insert some fake zero bits. */
    goto no_more_bytes;
}

/* OK, load c into get_buffer */
get_buffer = (get_buffer << 8) | c;
bits_left += 8;
} /* end while */
} else {
no_more_bytes:
/* We get here if we've read the marker that terminates the compressed
 * data segment. There should be enough bits in the buffer register
 * to satisfy the request; if so, no problem.
 */
if (nbits > bits_left) {
/* Uh-oh. Report corrupted data to user and stuff zeroes into
 * the data stream, so that we can produce some kind of image.
 * We use a nonvolatile flag to ensure that only one warning message
 * appears per data segment.
 */
    if (! cinfo->entropy->insufficient_data) {
        WARNMS(cinfo, JWRN_HIT_MARKER);
        cinfo->entropy->insufficient_data = TRUE;
    }
/* Fill the buffer with zero bits */
    get_buffer <= MIN_GET_BITS - bits_left;
    bits_left = MIN_GET_BITS;
}

/* Unload the local registers */
state->next_input_byte = next_input_byte;
state->bytes_in_buffer = bytes_in_buffer;
state->get_buffer = get_buffer;
state->bits_left = bits_left;

return TRUE;
}

/*
 * Out-of-line code for Huffman code decoding.
 * See jdhuft.h for info about usage.
 */

GLOBAL(int)
jpeg_huff_decode (bitread_working_state * state,
    register bit_buf_type get_buffer, register int bits_left,
    d_derived_tbl * htbl, int min_bits)

```

```

(
register int l = min_bits;
register INT32 code;

/* HUFF_DECODE has determined that the code is at least min_bits */
/* bits long, so fetch that many bits in one swoop. */

CHECK_BIT_BUFFER(*state, l, return -1);
code = GET_BITS(l);

/* Collect the rest of the Huffman code one bit at a time. */
/* This is per Figure F.16 in the JPEG spec. */

while (code > htbl->maxcode[l]) {
    code <= 1;
    CHECK_BIT_BUFFER(*state, 1, return -1);
    code |= GET_BITS(1);
    l++;
}

/* Unload the local registers */
state->get_buffer = get_buffer;
state->bits_left = bits_left;

/* With garbage input we may reach the sentinel value l = 17. */
if (l > 16) {
    WARNMS(state->cinfo, JWRN_HUFF_BAD_CODE);
    return 0;          /* fake a zero as the safest result */
}

return htbl->pub->huffval[ (int) (code + htbl->valoffset[l]) ];
)

```

Figure F.12: extend sign bit.

On some machines, a shift and add will be faster than a table lookup.

```

#ifdef AVOID_TABLES
#define HUFF_EXTEND(x,s) ((x) < (1<<((s)-1)) ? (x) + (((-1)<<(s)) + 1) : (x))
#else
#define HUFF_EXTEND(x,s) ((x) < extend_test[s] ? (x) + extend_offset[s] : (x))
static const int extend_test[16] = /* entry n is 2**(n-1) */
{ 0, 0x0001, 0x0002, 0x0004, 0x0008, 0x0010, 0x0020, 0x0040, 0x0080,
  0x0100, 0x0200, 0x0400, 0x0800, 0x1000, 0x2000, 0x4000 };
static const int extend_offset[16] = /* entry n is (-1 << n) + 1 */
{ 0, ((-1)<<1) + 1, ((-1)<<2) + 1, ((-1)<<3) + 1, ((-1)<<4) + 1,
  ((-1)<<5) + 1, ((-1)<<6) + 1, ((-1)<<7) + 1, ((-1)<<8) + 1,
  ((-1)<<9) + 1, ((-1)<<10) + 1, ((-1)<<11) + 1, ((-1)<<12) + 1,
  ((-1)<<13) + 1, ((-1)<<14) + 1, ((-1)<<15) + 1 };
#endif /* AVOID_TABLES */

```

```

/*
 * Check for a restart marker & resynchronize decoder.
 * Returns FALSE if must suspend.
 */

```

```

LOCAL(boolean)
process_restart (j_decompress_ptr cinfo)
{
    huff_entropy_ptr entropy = (huff_entropy_ptr) cinfo->entropy;
    int ci;

    /* Throw away any unused bits remaining in bit buffer; */
    /* include any full bytes in next_marker's count of discarded bytes */
    cinfo->marker->discarded_bytes += entropy->bitstate.bits_left / 8;
    entropy->bitstate.bits_left = 0;

    /* Advance past the RSTn marker */
    if (! (*cinfo->marker->read_restart_marker) (cinfo))
        return FALSE;
}

```



```

/* Re-initialize DC predictor to 0 */
for (ci = 0; ci < cinfo->blocks_in_scan; ci++)
    entropy->saved.last_dc_val[ci] = 0;

/* Reset restart counter */
entropy->restarts_to_go = cinfo->restart_interval;

/* Reset out-of-data flag, unless read_restart_marker left us smack up
 * against a marker. In that case we will end up treating the next data
 * segment as empty, and we can avoid producing bogus output pixels by
 * leaving the flag set.
 */
if (cinfo->unread_marker == 0)
    entropy->pub.insufficient_data = FALSE;

return TRUE;
}

/*
 * Decode and return one MCU's worth of Huffman-compressed coefficients.
 * The coefficients are reordered from zigzag order into natural array order,
 * but are not dequantized.
 *
 * The i'th block of the MCU is stored into the block pointed to by
 * MCU_data[i]. WE ASSUME THIS AREA HAS BEEN ZEROED BY THE CALLER.
 * (Wholesale zeroing is usually a little faster than retail...)
 *
 * Returns FALSE if data source requested suspension. In that case no
 * changes have been made to permanent state. (Exception: some output
 * coefficients may already have been assigned. This is harmless for
 * this module, since we'll just re-assign them on the next call.)
 */
METHODDEF(boolean)
decode_mcu (j_decompress_ptr cinfo, JBLOCKROW *MCU_data)
{
    huff_entropy_ptr entropy = (huff_entropy_ptr) cinfo->entropy;
    int blkcn;
    BITREAD_STATE_VARS;
    savable_state state;

    /* Process restart marker if needed; may have to suspend */
    if (cinfo->restart_interval) {
        if (entropy->restarts_to_go == 0)
            if (! process_restart(cinfo))
                return FALSE;
    }

    /* If we've run out of data, just leave the MCU set to zeroes.
     * This way, we return uniform gray for the remainder of the segment.
     */
    if (! entropy->pub.insufficient_data) {
        /* Load up working state */
        BITREAD_LOAD_STATE(cinfo, entropy->bitstate);
        ASSIGN_STATE(state, entropy->saved);

        /* Outer loop handles each block in the MCU */
        for (blkcn = 0; blkcn < cinfo->blocks_in_MCU; blkcn++) {
            JBLOCKROW block = MCU_data[blkcn];
            d_derived_tbl * dctbl = entropy->dc_cur_tbls[blkcn];
            d_derived_tbl * actbl = entropy->ac_cur_tbls[blkcn];
            register int s, k, r;

            /* Decode a single block's worth of coefficients */

            /* Section F.2.2.1: decode the DC coefficient difference */
            HUFF_DECODE(s, br_state, dctbl, return FALSE, label1);
            if (s) {
                CHECK_BIT_BUFFER(br_state, s, return FALSE);
                r = GET_BITS(s);
                s = HUFF_EXTEND(r, s);
            }

            if (entropy->dc_needed[blkcn]) {
                /* Convert DC difference to actual value, update last_dc_val */
                int ci = cinfo->MCU_membership[blkcn];

```

```

s += state.last_dc_val[ci];
state.last_dc_val[ci] = s;
/* Output the DC coefficient (assumes jpeg_natural_order[0] = 0) */
(*block)[0] = (JCOEF) s;
}

if (entropy->ac_needed[blkcn]) {

/* Section F.2.2.2: decode the AC coefficients */
/* Since zeroes are skipped, output area must be cleared beforehand */
for (k = 1; k < DCTSIZE2; k++) {
    HUFF_DECODE(s, br_state, actbl, return FALSE, label2);

    r = s >> 4;
    s &= 15;

    if (s) {
        k += r;
        CHECK_BIT_BUFFER(br_state, s, return FALSE);
        r = GET_BITS(s);
        s = HUFF_EXTEND(r, s);
        /* Output coefficient in natural (dezigzagged) order.
         * Note: the extra entries in jpeg_natural_order[] will save us
         * if k >= DCTSIZE2, which could happen if the data is corrupted.
         */
        (*block)[jpeg_natural_order[k]] = (JCOEF) s;
    } else {
        if (r != 15)
            break;
        k += 15;
    }
}

} else {

/* Section F.2.2.2: decode the AC coefficients */
/* In this path we just discard the values */
for (k = 1; k < DCTSIZE2; k++) {
    HUFF_DECODE(s, br_state, actbl, return FALSE, label3);

    r = s >> 4;
    s &= 15;

    if (s) {
        k += r;
        CHECK_BIT_BUFFER(br_state, s, return FALSE);
        DROP_BITS(s);
    } else {
        if (r != 15)
            break;
        k += 15;
    }
}

}

/* Completed MCU, so update state */
BITREAD_SAVE_STATE(cinfo, entropy->bitstate);
ASSIGN_STATE(entropy->saved, state);
}

/* Account for restart interval (no-op if not using restarts) */
entropy->restarts_to_go--;

return TRUE;
}

/*
 * Module initialization routine for Huffman entropy decoding.
 */

GLOBAL(void)
jinit_huff_decoder (j_decompress_ptr cinfo)
{
    huff_entropy_ptr entropy;
    int i;

    entropy = (huff_entropy_ptr)

```

```

(*cinfo->mem->alloc_small)((j_common_ptr) cinfo, JPOOL_IMAGE,
    SIZEOF(huff_entropy_decoder));
cinfo->entropy = (struct jpeg_entropy_decoder *) entropy;
entropy->pub.start_pass = start_pass_huff_decoder;
entropy->pub.decode_mcu = decode_mcu;

```

```

/* Mark tables unallocated */
for (i = 0; i < NUM_HUFF_TBLS; i++) {
    entropy->dc_derived_tbls[i] = entropy->ac_derived_tbls[i] = NULL;
}

```

00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

```

/*
 * jinput.c
 *
 * Copyright (C) 1991-1997, Thomas G. Lane.
 * This file is part of the Independent JPEG Group's software.
 * For conditions of distribution and use, see the accompanying README file.
 *
 * This file contains input control logic for the JPEG decompressor.
 * These routines are concerned with controlling the decompressor's input
 * processing (marker reading and coefficient decoding). The actual input
 * reading is done in jdmarker.c, jdhuft.c, and jdphuff.c.
 */

#define JPEG_INTERNALS
#include "jinclude.h"
#include "jpeglib.h"

/* Private state */

typedef struct {
  struct jpeg_input_controller pub; /* public fields */

  boolean inheaders; /* TRUE until first SOS is reached */
} my_input_controller;

typedef my_input_controller * my_inputctl_ptr;

/* Forward declarations */
METHODDEF(int) consume_markers JPP((j_decompress_ptr cinfo));

/*
 * Routines to calculate various quantities related to the size of the image.
 */
LOCAL(void)
initial_setup (j_decompress_ptr cinfo)
/* Called once, when first SOS marker is reached */
{
  int ci;
  jpeg_component_info *comp_ptr;

  /* Make sure image isn't bigger than I can handle */
  if ((long) cinfo->image_height > (long) JPEG_MAX_DIMENSION ||
      (long) cinfo->image_width > (long) JPEG_MAX_DIMENSION)
    ERREXIT1(cinfo, JERR_IMAGE_TOO_BIG, (unsigned int) JPEG_MAX_DIMENSION);

  /* For now, precision must match compiled-in value... */
  if (cinfo->data_precision != BITS_IN_JSAMPLE)
    ERREXIT1(cinfo, JERR_BAD_PRECISION, cinfo->data_precision);

  /* Check that number of components won't exceed internal array sizes */
  if (cinfo->num_components > MAX_COMPONENTS)
    ERREXIT2(cinfo, JERR_COMPONENT_COUNT, cinfo->num_components,
             MAX_COMPONENTS);

  /* Compute maximum sampling factors; check factor validity */
  cinfo->max_h_samp_factor = 1;
  cinfo->max_v_samp_factor = 1;
  for (ci = 0, comp_ptr = cinfo->comp_info; ci < cinfo->num_components;
       ci++, comp_ptr++) {
    if (comp_ptr->h_samp_factor <= 0 || comp_ptr->h_samp_factor > MAX_SAMP_FACTOR ||
        comp_ptr->v_samp_factor <= 0 || comp_ptr->v_samp_factor > MAX_SAMP_FACTOR)
      ERREXIT(cinfo, JERR_BAD_SAMPLING);
    cinfo->max_h_samp_factor = MAX(cinfo->max_h_samp_factor,
                                   comp_ptr->h_samp_factor);
    cinfo->max_v_samp_factor = MAX(cinfo->max_v_samp_factor,
                                   comp_ptr->v_samp_factor);
  }

  /* We initialize DCT_scaled_size and min_DCT_scaled_size to DCTSIZE.
   * In the full decompressor, this will be overridden by jdmaster.c;
   * but in the transcoder, jdmaster.c is not used, so we must do it here.
   */
  cinfo->min_DCT_scaled_size = DCTSIZE;

  /* Compute dimensions of components */
  for (ci = 0, comp_ptr = cinfo->comp_info; ci < cinfo->num_components;

```

```

        ci++, compptr++) {
    compptr->DCT_scaled_size = DCTSIZE;
    /* Size in DCT blocks */
    compptr->width_in_blocks = (JDIMENSION)
        jdiv_round_up((long) cinfo->image_width * (long) compptr->h_samp_factor,
            (long) (cinfo->max_h_samp_factor * DCTSIZE));
    compptr->height_in_blocks = (JDIMENSION)
        jdiv_round_up((long) cinfo->image_height * (long) compptr->v_samp_factor,
            (long) (cinfo->max_v_samp_factor * DCTSIZE));
    /* downsampled_width and downsampled_height will also be overridden by
     * jdmaster.c if we are doing full decomposition. The transcoder library
     * doesn't use these values, but the calling application might.
     */
    /* Size in samples */
    compptr->downsampled_width = (JDIMENSION)
        jdiv_round_up((long) cinfo->image_width * (long) compptr->h_samp_factor,
            (long) cinfo->max_h_samp_factor);
    compptr->downsampled_height = (JDIMENSION)
        jdiv_round_up((long) cinfo->image_height * (long) compptr->v_samp_factor,
            (long) cinfo->max_v_samp_factor);
    /* Mark component needed, until color conversion says otherwise */
    compptr->component_needed = TRUE;
    /* Mark no quantization table yet saved for component */
    compptr->quant_table = NULL;
}

/* Compute number of fully interleaved MCU rows. */
cinfo->total_iMCU_rows = (JDIMENSION)
    jdiv_round_up((long) cinfo->image_height,
        (long) (cinfo->max_v_samp_factor * DCTSIZE));

/* Decide whether file contains multiple scans */
if (cinfo->comps_in_scan < cinfo->num_components || cinfo->progressive_mode)
    cinfo->inputctl->has_multiple_scans = TRUE;
else
    cinfo->inputctl->has_multiple_scans = FALSE;
}

LOCAL(void)
per_scan_setup (j_decompress_ptr cinfo)
/* Do computations that are needed before processing a JPEG scan */
/* cinfo->comps_in_scan and cinfo->cur_comp_info[] were set from SOS marker */
{
    int ci, mcublks, tmp;
    jpeg_component_info *compptr;

    if (cinfo->comps_in_scan == 1) {
        /* Noninterleaved (single-component) scan */
        compptr = cinfo->cur_comp_info[0];

        /* Overall image size in MCUs */
        cinfo->MCUs_per_row = compptr->width_in_blocks;
        cinfo->MCU_rows_in_scan = compptr->height_in_blocks;

        /* For noninterleaved scan, always one block per MCU */
        compptr->MCU_width = 1;
        compptr->MCU_height = 1;
        compptr->MCU_blocks = 1;
        compptr->MCU_sample_width = compptr->DCT_scaled_size;
        compptr->last_col_width = 1;
        /* For noninterleaved scans, it is convenient to define last_row_height
         * as the number of block rows present in the last iMCU row.
         */
        tmp = (int) (compptr->height_in_blocks % compptr->v_samp_factor);
        if (tmp == 0) tmp = compptr->v_samp_factor;
        compptr->last_row_height = tmp;

        /* Prepare array describing MCU composition */
        cinfo->blocks_in_MCU = 1;
        cinfo->MCU_membership[0] = 0;
    } else {
        /* Interleaved (multi-component) scan */
        if (cinfo->comps_in_scan <= 0 || cinfo->comps_in_scan > MAX_COMPS_IN_SCAN)
            ERREXIT2(cinfo, JERR_COMPONENT_COUNT, cinfo->comps_in_scan,
                MAX_COMPS_IN_SCAN);
    }
}

```

```

/* Overall image size in MCUs */
cinfo->MCUs_per_row = (JDIMENSION)
    jdiv_round_up((long) cinfo->image_width,
        (long) (cinfo->max_h_samp_factor*DCTSIZE));
cinfo->MCU_rows_in_scan = (JDIMENSION)
    jdiv_round_up((long) cinfo->image_height,
        (long) (cinfo->max_v_samp_factor*DCTSIZE));

cinfo->blocks_in_MCU = 0;

for (ci = 0; ci < cinfo->comps_in_scan; ci++) {
    compptr = cinfo->cur_comp_info[ci];
    /* Sampling factors give # of blocks of component in each MCU */
    compptr->MCU_width = compptr->h_samp_factor;
    compptr->MCU_height = compptr->v_samp_factor;
    compptr->MCU_blocks = compptr->MCU_width * compptr->MCU_height;
    compptr->MCU_sample_width = compptr->MCU_width * compptr->DCT_scaled_size;
    /* Figure number of non-dummy blocks in last MCU column & row */
    tmp = (int) (compptr->width_in_blocks % compptr->MCU_width);
    if (tmp == 0) tmp = compptr->MCU_width;
    compptr->last_col_width = tmp;
    tmp = (int) (compptr->height_in_blocks % compptr->MCU_height);
    if (tmp == 0) tmp = compptr->MCU_height;
    compptr->last_row_height = tmp;
    /* Prepare array describing MCU composition */
    mcublk = compptr->MCU_blocks;
    if (cinfo->blocks_in_MCU + mcublk > D_MAX_BLOCKS_IN_MCU)
        ERREXIT(cinfo, JERR_BAD_MCU_SIZE);
    while (mcublk-- > 0) {
        cinfo->MCU_membership[cinfo->blocks_in_MCU++] = ci;
    }
}

```

Save away a copy of the Q-table referenced by each component present in the current scan, unless already saved during a prior scan.

In a multiple-scan JPEG file, the encoder could assign different components the same Q-table slot number, but change table definitions between scans so that each component uses a different Q-table. (The IJG encoder is not currently capable of doing this, but other encoders might.) Since we want to be able to dequantize all the components at the end of the file, this means that we have to save away the table actually used for each component. We do this by copying the table at the start of the first scan containing the component.

The JPEG spec prohibits the encoder from changing the contents of a Q-table slot between scans of a component using that slot. If the encoder does so anyway, this decoder will simply use the Q-table values that were current at the start of the first scan for the component.

\* The decompressor output side looks only at the saved quant tables,  
 \* not at the current Q-table slots.  
 \*/

```

LOCAL(void)
latch_quant_tables (j_decompress_ptr cinfo)
{
    int ci, qtblno;
    jpeg_component_info *compptr;
    JQUANT_TBL * qtbl;

    for (ci = 0; ci < cinfo->comps_in_scan; ci++) {
        compptr = cinfo->cur_comp_info[ci];
        /* No work if we already saved Q-table for this component */
        if (compptr->quant_table != NULL)
            continue;
        /* Make sure specified quantization table is present */
        qtblno = compptr->quant_tbl_no;
        if (qtblno < 0 || qtblno >= NUM_QUANT_TBLS ||
            cinfo->quant_tbl_ptrs[qtblno] == NULL)
            ERREXIT1(cinfo, JERR_NO_QUANT_TABLE, qtblno);
        /* OK, save away the quantization table */
        qtbl = (JQUANT_TBL *)
            (*cinfo->mem->alloc_small) ((j_common_ptr) cinfo, JPOOL_IMAGE,
                SIZEOF(JQUANT_TBL));
        MEMCOPY(qtbl, cinfo->quant_tbl_ptrs[qtblno], SIZEOF(JQUANT_TBL));
    }
}

```

```

    compptr->quant_table = qt;
}

/*
 * Initialize the input modules to read a scan of compressed data.
 * The first call to this is done by jdmaster.c after initializing
 * the entire decompressor (during jpeg_start_decompress).
 * Subsequent calls come from consume_markers, below.
 */

METHODDEF(void)
start_input_pass (j_decompress_ptr cinfo)
{
    per_scan_setup(cinfo);
    latch_quant_tables(cinfo);
    (*cinfo->entropy->start_pass) (cinfo);
    (*cinfo->coef->start_input_pass) (cinfo);
    cinfo->inputctl->consume_input = cinfo->coef->consume_data;
}

/*
 * Finish up after inputting a compressed-data scan.
 * This is called by the coefficient controller after it's read all
 * the expected data of the scan.
 */

METHODDEF(void)
finish_input_pass (j_decompress_ptr cinfo)
{
    cinfo->inputctl->consume_input = consume_markers;
}

/*
 * Read JPEG markers before, between, or after compressed-data scans.
 * Change state as necessary when a new scan is reached.
 * Return value is JPEG_SUSPENDED, JPEG_REACHED_SOS, or JPEG_REACHED_EOI.
 *
 * The consume_input method pointer points either here or to the
 * coefficient controller's consume_data routine, depending on whether
 * we are reading a compressed data segment or inter-segment markers.
 */

METHODDEF(int)
consume_markers (j_decompress_ptr cinfo)
{
    my_inputctl_ptr inputctl = (my_inputctl_ptr) cinfo->inputctl;
    int val;

    if (inputctl->pub.eoi_reached) /* After hitting EOI, read no further */
        return JPEG_REACHED_EOI;

    val = (*cinfo->marker->read_markers) (cinfo);

    switch (val) {
        case JPEG_REACHED_SOS: /* Found SOS */
            if (inputctl->inheaders) { /* 1st SOS */
                initial_setup(cinfo);
                inputctl->inheaders = FALSE;
                /* Note: start_input_pass must be called by jdmaster.c
                 * before any more input can be consumed. jdapimin.c is
                 * responsible for enforcing this sequencing.
                 */
            } else { /* 2nd or later SOS marker */
                if (!inputctl->pub.has_multiple_scans)
                    ERREXIT(cinfo, JERR_EOI_EXPECTED); /* Oops, I wasn't expecting this! */
                start_input_pass(cinfo);
            }
            break;
        case JPEG_REACHED_EOI: /* Found EOI */
            inputctl->pub.eoi_reached = TRUE;
            if (inputctl->inheaders) { /* Tables-only datastream, apparently */
                if (cinfo->marker->saw_SOF)
                    ERREXIT(cinfo, JERR_SOF_NO_SOS);
            } else {
                /* Prevent infinite loop in coef ctlr's decompress_data routine
                 * if user set output_scan_number larger than number of scans.

```

```

        */
        if (cinfo->output_scan_number > cinfo->input_scan_number)
            cinfo->output_scan_number = cinfo->input_scan_number;
    }
    break;
case JPEG_SUSPENDED:
    break;
}

return val;
}

/*
 * Reset state to begin a fresh datastream.
 */

METHODDEF(void)
reset_input_controller (j_decompress_ptr cinfo)
{
    my_inputctl_ptr inputctl = (my_inputctl_ptr) cinfo->inputctl;

    inputctl->pub.consume_input = consume_markers;
    inputctl->pub.has_multiple_scans = FALSE; /* "unknown" would be better */
    inputctl->pub.eoi_reached = FALSE;
    inputctl->inheaders = TRUE;
    /* Reset other modules */
    (*cinfo->err->reset_error_mgr) ((j_common_ptr) cinfo);
    (*cinfo->marker->reset_marker_reader) (cinfo);
    /* Reset progression state -- would be cleaner if entropy decoder did this */
    cinfo->coef_bits = NULL;
}

/*
 * Initialize the input controller module.
 * This is called only once, when the decompression object is created.
 */

GLOBAL(void)
jinit_input_controller (j_decompress_ptr cinfo)
{
    my_inputctl_ptr inputctl;

    /* Create subobject in permanent pool */
    inputctl = (my_inputctl_ptr)
        (*cinfo->mem->alloc_small) ((j_common_ptr) cinfo, JPOOL_PERMANENT,
            SIZEOF(my_input_controller));
    cinfo->inputctl = (struct jpeg_input_controller *) inputctl;
    /* Initialize method pointers */
    inputctl->pub.consume_input = consume_markers;
    inputctl->pub.reset_input_controller = reset_input_controller;
    inputctl->pub.start_input_pass = start_input_pass;
    inputctl->pub.finish_input_pass = finish_input_pass;
    /* Initialize state: can't use reset_input_controller since we don't
     * want to try to reset other modules yet.
     */
    inputctl->pub.has_multiple_scans = FALSE; /* "unknown" would be better */
    inputctl->pub.eoi_reached = FALSE;
    inputctl->inheaders = TRUE;
}

```



```

/*
 * jdmaint.c
 *
 * Copyright (C) 1994-1996, Thomas G. Lane.
 * This file is part of the Independent JPEG Group's software.
 * For conditions of distribution and use, see the accompanying README file.
 *
 * This file contains the main buffer controller for decompression.
 * The main buffer lies between the JPEG decompressor proper and the
 * post-processor; it holds downsampled data in the JPEG colorspace.
 *
 * Note that this code is bypassed in raw-data mode, since the application
 * supplies the equivalent of the main buffer in that case.
 */

#define JPEG_INTERNALS
#include "jinclude.h"
#include "jpeglib.h"

/*
 * In the current system design, the main buffer need never be a full-image
 * buffer; any full-height buffers will be found inside the coefficient or
 * postprocessing controllers. Nonetheless, the main controller is not
 * trivial. Its responsibility is to provide context rows for upsampling/
 * rescaling, and doing this in an efficient fashion is a bit tricky.
 *
 * Postprocessor input data is counted in "row groups". A row group
 * is defined to be (v_samp_factor * DCT_scaled_size / min_DCT_scaled_size)
 * sample rows of each component. (We require DCT_scaled_size values to be
 * chosen such that these numbers are integers. In practice DCT_scaled_size
 * values will likely be powers of two, so we actually have the stronger
 * condition that DCT_scaled_size / min_DCT_scaled_size is an integer.)
 * Upsampling will typically produce max_v_samp_factor pixel rows from each
 * row group (times any additional scale factor that the upsampler is
 * applying).
 *
 * The coefficient controller will deliver data to us one iMCU row at a time;
 * each iMCU row contains v_samp_factor * DCT_scaled_size sample rows, or
 * exactly min_DCT_scaled_size row groups. (This amount of data corresponds
 * to one row of MCUs when the image is fully interleaved.) Note that the
 * number of sample rows varies across components, but the number of row
 * groups does not. Some garbage sample rows may be included in the last iMCU
 * row at the bottom of the image.
 *
 * Depending on the vertical scaling algorithm used, the upsampler may need
 * access to the sample row(s) above and below its current input row group.
 * The upsampler is required to set need_context_rows TRUE at global selection
 * time if so. When need_context_rows is FALSE, this controller can simply
 * obtain one iMCU row at a time from the coefficient controller and dole it
 * out as row groups to the postprocessor.
 *
 * When need_context_rows is TRUE, this controller guarantees that the buffer
 * passed to postprocessing contains at least one row group's worth of samples
 * above and below the row group(s) being processed. Note that the context
 * rows "above" the first passed row group appear at negative row offsets in
 * the passed buffer. At the top and bottom of the image, the required
 * context rows are manufactured by duplicating the first or last real sample
 * row; this avoids having special cases in the upsampling inner loops.
 *
 * The amount of context is fixed at one row group just because that's a
 * convenient number for this controller to work with. The existing
 * upsamplers really only need one sample row of context. An upsampler
 * supporting arbitrary output rescaling might wish for more than one row
 * group of context when shrinking the image; tough, we don't handle that.
 * (This is justified by the assumption that downsizing will be handled mostly
 * by adjusting the DCT_scaled_size values, so that the actual scale factor at
 * the upsample step needn't be much less than one.)
 *
 * To provide the desired context, we have to retain the last two row groups
 * of one iMCU row while reading in the next iMCU row. (The last row group
 * can't be processed until we have another row group for its below-context,
 * and so we have to save the next-to-last group too for its above-context.)
 * We could do this most simply by copying data around in our buffer, but
 * that'd be very slow. We can avoid copying any data by creating a rather
 * strange pointer structure. Here's how it works. We allocate a workspace
 * consisting of M+2 row groups (where M = min_DCT_scaled_size is the number
 * of row groups per iMCU row). We create two sets of redundant pointers to
 * the workspace. Labeling the physical row groups 0 to M+1, the synthesized
 * pointer lists look like this:

```

```

*           M+1           M-1
* master pointer --> 0   master pointer --> 0
*           1           1
*           ...           ...
*           M-3           M-3
*           M-2           M
*           M-1           M+1
*           M             M-2
*           M+1           M-1
*           0             0
* We read alternate iMCU rows using each master pointer; thus the last two
* row groups of the previous iMCU row remain un-overwritten in the workspace.
* The pointer lists are set up so that the required context rows appear to
* be adjacent to the proper places when we pass the pointer lists to the
* upsampler.
*
* The above pictures describe the normal state of the pointer lists.
* At top and bottom of the image, we diddle the pointer lists to duplicate
* the first or last sample row as necessary (this is cheaper than copying
* sample rows around).
*
* This scheme breaks down if M < 2, ie, min_DCT_scaled_size is 1. In that
* situation each iMCU row provides only one row group so the buffering logic
* must be different (eg, we must read two iMCU rows before we can emit the
* first row group). For now, we simply do not support providing context
* rows when min_DCT_scaled_size is 1. That combination seems unlikely to
* be worth providing --- if someone wants a 1/8th-size preview, they probably
* want it quick and dirty, so a context-free upsampler is sufficient.
*/

```

```

/* Private buffer controller object */

```

```

typedef struct {
    struct jpeg_d_main_controller pub; /* public fields */

    /* Pointer to allocated workspace (M or M+2 row groups). */
    JSAMPARRAY buffer[MAX_COMPONENTS];

    boolean buffer_full; /* Have we gotten an iMCU row from decoder? */
    JDIMENSION rowgroup_ctr; /* counts row groups output to postprocessor */

    /* Remaining fields are only used in the context case. */

    /* These are the master pointers to the funny-order pointer lists. */
    JSAMPIMAGE xbuffer[2]; /* pointers to weird pointer lists */

    int whichptr; /* indicates which pointer set is now in use */
    int context_state; /* process_data state machine status */
    JDIMENSION rowgroups_avail; /* row groups available to postprocessor */
    JDIMENSION iMCU_row_ctr; /* counts iMCU rows to detect image top/bot */
    my_main_controller;

    typedef my_main_controller * my_main_ptr;

    /* context_state values: */
#define CTX_PREPARE_FOR_IMCU 0 /* need to prepare for MCU row */
#define CTX_PROCESS_IMCU 1 /* feeding iMCU to postprocessor */
#define CTX_POSTPONED_ROW 2 /* feeding postponed row group */

```

```

/* Forward declarations */
METHODDEF(void) process_data_simple_main
    JPP((j_decompress_ptr cinfo, JSAMPARRAY output_buf,
        JDIMENSION *out_row_ctr, JDIMENSION out_rows_avail));
METHODDEF(void) process_data_context_main
    JPP((j_decompress_ptr cinfo, JSAMPARRAY output_buf,
        JDIMENSION *out_row_ctr, JDIMENSION out_rows_avail));
#ifdef QUANT_2PASS_SUPPORTED
METHODDEF(void) process_data_crank_post
    JPP((j_decompress_ptr cinfo, JSAMPARRAY output_buf,
        JDIMENSION *out_row_ctr, JDIMENSION out_rows_avail));
#endif

```

```

LOCAL(void)
alloc_funny_pointers (j_decompress_ptr cinfo)
/* Allocate space for the funny pointer lists.
 * This is done only once, not once per pass.
 */

```

```

{
my_main_ptr main = (my_main_ptr) cinfo->main;
int ci, rgroup;
int M = cinfo->min_DCT_scaled_size;
jpeg_component_info *comp_ptr;
JSAMPARRAY xbuf;

/* Get top-level space for component array pointers.
 * We alloc both arrays with one call to save a few cycles.
 */
main->xbuffer[0] = (JSAMPIMAGE)
    (*cinfo->mem->alloc_small) ((j_common_ptr) cinfo, JPOOL_IMAGE,
        cinfo->num_components * 2 * SIZEOF(JSAMPARRAY));
main->xbuffer[1] = main->xbuffer[0] + cinfo->num_components;

for (ci = 0, comp_ptr = cinfo->comp_info; ci < cinfo->num_components;
    ci++, comp_ptr++) {
    rgroup = (comp_ptr->v_samp_factor * comp_ptr->DCT_scaled_size) /
        cinfo->min_DCT_scaled_size; /* height of a row group of component */
    /* Get space for pointer lists --- M+4 row groups in each list.
     * We alloc both pointer lists with one call to save a few cycles.
     */
    xbuf = (JSAMPARRAY)
        (*cinfo->mem->alloc_small) ((j_common_ptr) cinfo, JPOOL_IMAGE,
            2 * (rgroup * (M + 4)) * SIZEOF(JSAMPARRAY));
    xbuf += rgroup; /* want one row group at negative offsets */
    main->xbuffer[0][ci] = xbuf;
    xbuf += rgroup * (M + 4);
    main->xbuffer[1][ci] = xbuf;
}
}

LOCAL(void)
make_funny_pointers (j_decompress_ptr cinfo)
/* Create the funny pointer lists discussed in the comments above.
 * The actual workspace is already allocated (in main->buffer),
 * and the space for the pointer lists is allocated too.
 * This routine just fills in the curiously ordered lists.
 * This will be repeated at the beginning of each pass.
 */
{
my_main_ptr main = (my_main_ptr) cinfo->main;
int ci, i, rgroup;
int M = cinfo->min_DCT_scaled_size;
jpeg_component_info *comp_ptr;
JSAMPARRAY buf, xbuf0, xbuf1;

for (ci = 0, comp_ptr = cinfo->comp_info; ci < cinfo->num_components;
    ci++, comp_ptr++) {
    rgroup = (comp_ptr->v_samp_factor * comp_ptr->DCT_scaled_size) /
        cinfo->min_DCT_scaled_size; /* height of a row group of component */
    xbuf0 = main->xbuffer[0][ci];
    xbuf1 = main->xbuffer[1][ci];
    /* First copy the workspace pointers as-is */
    buf = main->buffer[ci];
    for (i = 0; i < rgroup * (M + 2); i++) {
        xbuf0[i] = xbuf1[i] = buf[i];
    }
    /* In the second list, put the last four row groups in swapped order */
    for (i = 0; i < rgroup * 2; i++) {
        xbuf1[rgroup*(M-2) + i] = buf[rgroup*M + i];
        xbuf1[rgroup*M + i] = buf[rgroup*(M-2) + i];
    }
    /* The wraparound pointers at top and bottom will be filled later
     * (see set_wraparound_pointers, below). Initially we want the "above"
     * pointers to duplicate the first actual data line. This only needs
     * to happen in xbuffer[0].
     */
    for (i = 0; i < rgroup; i++) {
        xbuf0[i - rgroup] = xbuf0[0];
    }
}
}

LOCAL(void)
set_wraparound_pointers (j_decompress_ptr cinfo)
/* Set up the "wraparound" pointers at top and bottom of the pointer lists.
 * This changes the pointer list state from top-of-image to the normal state.
 */

```

```

*/
{
    my_main_ptr main = (my_main_ptr) cinfo->main;
    int ci, i, rgroup;
    int M = cinfo->min_DCT_scaled_size;
    jpeg_component_info *comp_ptr;
    JSAMPARRAY xbuf0, xbuf1;

    for (ci = 0, comp_ptr = cinfo->comp_info; ci < cinfo->num_components;
         ci++, comp_ptr++) {
        rgroup = (comp_ptr->v_samp_factor * comp_ptr->DCT_scaled_size) /
            cinfo->min_DCT_scaled_size; /* height of a row group of component */
        xbuf0 = main->xbuffer[0][ci];
        xbuf1 = main->xbuffer[1][ci];
        for (i = 0; i < rgroup; i++) {
            xbuf0[i - rgroup] = xbuf0[rgroup*(M+1) + i];
            xbuf1[i - rgroup] = xbuf1[rgroup*(M+1) + i];
            xbuf0[rgroup*(M+2) + i] = xbuf0[i];
            xbuf1[rgroup*(M+2) + i] = xbuf1[i];
        }
    }
}

LOCAL(void)
set_bottom_pointers (j_decompress_ptr cinfo)
/* Change the pointer lists to duplicate the last sample row at the bottom
 * of the image.  which_ptr indicates which xbuffer holds the final iMCU row.
 * Also sets rowgroups_avail to indicate number of nondummy row groups in row.
 */
{
    my_main_ptr main = (my_main_ptr) cinfo->main;
    int ci, i, rgroup, iMCUheight, rows_left;
    jpeg_component_info *comp_ptr;
    JSAMPARRAY xbuf;

    for (ci = 0, comp_ptr = cinfo->comp_info; ci < cinfo->num_components;
         ci++, comp_ptr++) {
        /* Count sample rows in one iMCU row and in one row group */
        iMCUheight = comp_ptr->v_samp_factor * comp_ptr->DCT_scaled_size;
        rgroup = iMCUheight / cinfo->min_DCT_scaled_size;
        /* Count nondummy sample rows remaining for this component */
        rows_left = (int) (comp_ptr->downsampled_height % (JDIMENSION) iMCUheight);
        if (rows_left == 0) rows_left = iMCUheight;
        /* Count nondummy row groups.  Should get same answer for each component,
         * so we need only do it once.
         */
        if (ci == 0) {
            main->rowgroups_avail = (JDIMENSION) ((rows_left-1) / rgroup + 1);
        }
        /* Duplicate the last real sample row rgroup*2 times; this pads out the
         * last partial rowgroup and ensures at least one full rowgroup of context.
         */
        xbuf = main->xbuffer[main->which_ptr][ci];
        for (i = 0; i < rgroup * 2; i++) {
            xbuf[rows_left + i] = xbuf[rows_left-1];
        }
    }
}

/*
 * Initialize for a processing pass.
 */

METHODDEF(void)
start_pass_main (j_decompress_ptr cinfo, J_BUF_MODE pass_mode)
{
    my_main_ptr main = (my_main_ptr) cinfo->main;

    switch (pass_mode) {
        case JBUF_PASS_THRU:
            if (cinfo->upsample->need_context_rows) {
                main->pub.process_data = process_data_context_main;
                make_funny_pointers(cinfo); /* Create the xbuffer[] lists */
                main->which_ptr = 0; /* Read first iMCU row into xbuffer[0] */
                main->context_state = CTX_PREPARE_FOR_IMCU;
                main->iMCU_row_ctr = 0;
            } else {
                /* Simple case with no context needed */
            }
    }
}

```

```

    main->pub.process_data = process_data_simple_main;
}
main->buffer_full = FALSE; /* Mark buffer empty */
main->rowgroup_ctr = 0;
break;
#ifdef QUANT_2PASS_SUPPORTED
case JBUF_CRANK_DEST:
    /* For last pass of 2-pass quantization, just crank the postprocessor */
    main->pub.process_data = process_data_crank_post;
    break;
#endif
default:
    ERREXIT(cinfo, JERR_BAD_BUFFER_MODE);
    break;
}
}

/*
 * Process some data.
 * This handles the simple case where no context is required.
 */

METHODDEF(void)
process_data_simple_main (j_decompress_ptr cinfo,
                          JSAMPARRAY output_buf, JDIMENSION *out_row_ctr,
                          JDIMENSION out_rows_avail)
{
    my_main_ptr main = (my_main_ptr) cinfo->main;
    JDIMENSION rowgroups_avail;

    /* Read input data if we haven't filled the main buffer yet */
    if (! main->buffer_full) {
        if (! (*cinfo->coef->decompress_data) (cinfo, main->buffer))
            return; /* suspension forced, can do nothing more */
        main->buffer_full = TRUE; /* OK, we have an iMCU row to work with */
    }

    /* There are always min_DCT_scaled_size row groups in an iMCU row. */
    rowgroups_avail = (JDIMENSION) cinfo->min_DCT_scaled_size;
    /* Note: at the bottom of the image, we may pass extra garbage row groups
     * to the postprocessor. The postprocessor has to check for bottom
     * of image anyway (at row resolution), so no point in us doing it too.
     */

    /* Feed the postprocessor */
    (*cinfo->post->post_process_data) (cinfo, main->buffer,
                                       &main->rowgroup_ctr, rowgroups_avail,
                                       output_buf, out_row_ctr, out_rows_avail);

    /* Has postprocessor consumed all the data yet? If so, mark buffer empty */
    if (main->rowgroup_ctr >= rowgroups_avail) {
        main->buffer_full = FALSE;
        main->rowgroup_ctr = 0;
    }
}

/*
 * Process some data.
 * This handles the case where context rows must be provided.
 */

METHODDEF(void)
process_data_context_main (j_decompress_ptr cinfo,
                           JSAMPARRAY output_buf, JDIMENSION *out_row_ctr,
                           JDIMENSION out_rows_avail)
{
    my_main_ptr main = (my_main_ptr) cinfo->main;

    /* Read input data if we haven't filled the main buffer yet */
    if (! main->buffer_full) {
        if (! (*cinfo->coef->decompress_data) (cinfo,
                                              main->xbuffer[main->whichptr]))
            return; /* suspension forced, can do nothing more */
        main->buffer_full = TRUE; /* OK, we have an iMCU row to work with */
        main->iMCU_row_ctr++; /* count rows received */
    }

    /* Postprocessor typically will not swallow all the input data it is handed

```

```

* in one call (due to filling the output buffer first). Must be prepared
* to exit and restart. The switch lets us keep track of how we've got.
* Note that each case falls through to the next on successful completion.
*/
switch (main->context_state) {
case CTX_POSTPONED_ROW:
/* Call postprocessor using previously set pointers for postponed row */
(*cinfo->post->post_process_data) (cinfo, main->xbuffer[main->whichptr],
&main->rowgroup_ctr, main->rowgroups_avail,
output_buf, out_row_ctr, out_rows_avail);
if (main->rowgroup_ctr < main->rowgroups_avail)
return; /* Need to suspend */
main->context_state = CTX_PREPARE_FOR_IMCU;
if (*out_row_ctr >= out_rows_avail)
return; /* Postprocessor exactly filled output buf */
/*FALLTHROUGH*/
case CTX_PREPARE_FOR_IMCU:
/* Prepare to process first M-1 row groups of this iMCU row */
main->rowgroup_ctr = 0;
main->rowgroups_avail = (JDIMENSION) (cinfo->min_DCT_scaled_size - 1);
/* Check for bottom of image: if so, tweak pointers to "duplicate"
* the last sample row, and adjust rowgroups_avail to ignore padding rows.
*/
if (main->iMCU_row_ctr == cinfo->total_iMCU_rows)
set_bottom_pointers(cinfo);
main->context_state = CTX_PROCESS_IMCU;
/*FALLTHROUGH*/
case CTX_PROCESS_IMCU:
/* Call postprocessor using previously set pointers */
(*cinfo->post->post_process_data) (cinfo, main->xbuffer[main->whichptr],
&main->rowgroup_ctr, main->rowgroups_avail,
output_buf, out_row_ctr, out_rows_avail);
if (main->rowgroup_ctr < main->rowgroups_avail)
return; /* Need to suspend */
/* After the first iMCU, change wraparound pointers to normal state */
if (main->iMCU_row_ctr == 1)
set_wraparound_pointers(cinfo);
/* Prepare to load new iMCU row using other xbuffer list */
main->whichptr ^= 1; /* 0=>1 or 1=>0 */
main->buffer_full = FALSE;
/* Still need to process last row group of this iMCU row, */
/* which is saved at index M+1 of the other xbuffer */
main->rowgroup_ctr = (JDIMENSION) (cinfo->min_DCT_scaled_size + 1);
main->rowgroups_avail = (JDIMENSION) (cinfo->min_DCT_scaled_size + 2);
main->context_state = CTX_POSTPONED_ROW;
}
}

Process some data.
Final pass of two-pass quantization: just call the postprocessor.
Source data will be the postprocessor controller's internal buffer.
*/

#ifdef QUANT_2PASS_SUPPORTED

METHODDEF(void)
process_data_crank_post (j_decompress_ptr cinfo,
JSAMPARRAY output_buf, JDIMENSION *out_row_ctr,
JDIMENSION out_rows_avail)
{
(*cinfo->post->post_process_data) (cinfo, (JSAMPIMAGE) NULL,
(JDIMENSION *) NULL, (JDIMENSION) 0,
output_buf, out_row_ctr, out_rows_avail);
}

#endif /* QUANT_2PASS_SUPPORTED */

/*
* Initialize main buffer controller.
*/

GLOBAL(void)
jinit_d_main_controller (j_decompress_ptr cinfo, boolean need_full_buffer)
{
my_main_ptr main;
int ci, rgroup, ngroups;
jpeg_component_info *comp_ptr;

```

```

main = (my_main_ptr)
(*cinfo->mem->alloc_small((j_common_ptr) cinfo, JPOOL_IMAGE,
    SIZEOF(my_main_controller)));
cinfo->main = (struct jpeg_d_main_controller *) main;
main->pub.start_pass = start_pass_main;

if (need_full_buffer) /* shouldn't happen */
    ERREXIT(cinfo, JERR_BAD_BUFFER_MODE);

/* Allocate the workspace.
 * ngroups is the number of row groups we need.
 */
if (cinfo->upsample->need_context_rows) {
    if (cinfo->min_DCT_scaled_size < 2) /* unsupported, see comments above */
        ERREXIT(cinfo, JERR_NOTIMPL);
    alloc_funny_pointers(cinfo); /* Alloc space for xbuffer[] lists */
    ngroups = cinfo->min_DCT_scaled_size + 2;
} else {
    ngroups = cinfo->min_DCT_scaled_size;
}

for (ci = 0, compptr = cinfo->comp_info; ci < cinfo->num_components;
    ci++, compptr++) {
    rgroup = (compptr->v_samp_factor * compptr->DCT_scaled_size) /
        cinfo->min_DCT_scaled_size; /* height of a row group of component */
    main->buffer[ci] = (*cinfo->mem->alloc_sarray)
        ((j_common_ptr) cinfo, JPOOL_IMAGE,
        compptr->width_in_blocks * compptr->DCT_scaled_size,
        (JDIMENSION) (rgroup * ngroups));
}
}

```

```

/*
 * jdmarker.c
 *
 * Copyright (C) 1991-1998, Thomas G. Lane.
 * This file is part of the Independent JPEG Group's software.
 * For conditions of distribution and use, see the accompanying README file.
 *
 * This file contains routines to decode JPEG datastream markers.
 * Most of the complexity arises from our desire to support input
 * suspension: if not all of the data for a marker is available,
 * we must exit back to the application.  On resumption, we reprocess
 * the marker.
 */

```

```

#define JPEG_INTERNALS
#include "jinclude.h"
#include "jpeglib.h"

```

```

typedef enum {          /* JPEG marker codes */

```

```

  M_SOF0 = 0xc0,
  M_SOF1 = 0xc1,
  M_SOF2 = 0xc2,
  M_SOF3 = 0xc3,

```

```

  M_SOF5 = 0xc5,
  M_SOF6 = 0xc6,
  M_SOF7 = 0xc7,

```

```

  M_JPG = 0xc8,
  M_SOF9 = 0xc9,
  M_SOF10 = 0xca,
  M_SOF11 = 0xcb,

```

```

  M_SOF13 = 0xcd,
  M_SOF14 = 0xce,
  M_SOF15 = 0xcf,

```

```

  M_DHT = 0xc4,

```

```

  M_DAC = 0xcc,

```

```

  M_RST0 = 0xd0,
  M_RST1 = 0xd1,
  M_RST2 = 0xd2,
  M_RST3 = 0xd3,
  M_RST4 = 0xd4,
  M_RST5 = 0xd5,
  M_RST6 = 0xd6,
  M_RST7 = 0xd7,

```

```

  M_SOI = 0xd8,

```

```

  M_EOI = 0xd9,

```

```

  M_SOS = 0xda,

```

```

  M_DQT = 0xdb,

```

```

  M_DNL = 0xdc,

```

```

  M_DRI = 0xdd,

```

```

  M_DHP = 0xde,

```

```

  M_EXP = 0xdf,

```

```

  M_APP0 = 0xe0,

```

```

  M_APP1 = 0xe1,

```

```

  M_APP2 = 0xe2,

```

```

  M_APP3 = 0xe3,

```

```

  M_APP4 = 0xe4,

```

```

  M_APP5 = 0xe5,

```

```

  M_APP6 = 0xe6,

```

```

  M_APP7 = 0xe7,

```

```

  M_APP8 = 0xe8,

```

```

  M_APP9 = 0xe9,

```

```

  M_APP10 = 0xea,

```

```

  M_APP11 = 0xeb,

```

```

  M_APP12 = 0xec,

```

```

  M_APP13 = 0xed,

```

```

  M_APP14 = 0xee,

```

```

  M_APP15 = 0xef,

```

```

  M_JPG0 = 0xf0,

```

```

  M_JPG13 = 0xfd,

```

```

  M_COM = 0xfe,

```



```

M_TEM = 0x01,
M_ERROR = 0x100
} JPEG_MARKER;

/* Private state */

typedef struct {
    struct jpeg_marker_reader pub; /* public fields */

    /* Application-overridable marker processing methods */
    jpeg_marker_parser_method process_COM;
    jpeg_marker_parser_method process_APPn[16];

    /* Limit on marker data length to save for each marker type */
    unsigned int length_limit_COM;
    unsigned int length_limit_APPn[16];

    /* Status of COM/AppN marker saving */
    jpeg_saved_marker_ptr cur_marker; /* NULL if not processing a marker */
    unsigned int bytes_read; /* data bytes read so far in marker */
    /* Note: cur_marker is not linked into marker_list until it's all read. */
} my_marker_reader;

typedef my_marker_reader * my_marker_ptr;

/*
 * Macros for fetching data from the data source module.
 *
 * At all times, cinfo->src->next_input_byte and ->bytes_in_buffer reflect
 * the current restart point; we update them only when we have reached a
 * suitable place to restart if a suspension occurs.
 */

/* Declare and initialize local copies of input pointer/count */
#define INPUT_VARS(cinfo) \
    struct jpeg_source_mgr * datasrc = (cinfo)->src; \
    const JOCTET * next_input_byte = datasrc->next_input_byte; \
    size_t bytes_in_buffer = datasrc->bytes_in_buffer

/* Unload the local copies --- do this only at a restart boundary */
#define INPUT_SYNC(cinfo) \
    ( datasrc->next_input_byte = next_input_byte, \
      datasrc->bytes_in_buffer = bytes_in_buffer )

/* Reload the local copies --- used only in MAKE_BYTE_AVAIL */
#define INPUT_RELOAD(cinfo) \
    ( next_input_byte = datasrc->next_input_byte, \
      bytes_in_buffer = datasrc->bytes_in_buffer )

/* Internal macro for INPUT_BYTE and INPUT_2BYTES: make a byte available.
 * Note we do *not* do INPUT_SYNC before calling fill_input_buffer,
 * but we must reload the local copies after a successful fill.
 */
#define MAKE_BYTE_AVAIL(cinfo,action) \
    if (bytes_in_buffer == 0) { \
        if (! (*datasrc->fill_input_buffer) (cinfo)) \
            { action; } \
        INPUT_RELOAD(cinfo); \
    }

/* Read a byte into variable V.
 * If must suspend, take the specified action (typically "return FALSE").
 */
#define INPUT_BYTE(cinfo,V,action) \
    MAKESTMT( MAKE_BYTE_AVAIL(cinfo,action); \
              bytes_in_buffer--; \
              V = GETJOCTET(*next_input_byte++); )

/* As above, but read two bytes interpreted as an unsigned 16-bit integer.
 * V should be declared unsigned int or perhaps INT32.
 */
#define INPUT_2BYTES(cinfo,V,action) \
    MAKESTMT( MAKE_BYTE_AVAIL(cinfo,action); \
              bytes_in_buffer--; \
              V = ((unsigned int) GETJOCTET(*next_input_byte++)) << 8; \
              MAKE_BYTE_AVAIL(cinfo,action); \

```

```

bytes_in_buffer--;
V += GETJOCTET(*next_input_byte++); )

```

```

/*
 * Routines to process JPEG markers.
 *
 * Entry condition: JPEG marker itself has been read and its code saved
 *   in cinfo->unread_marker; input restart point is just after the marker.
 *
 * Exit: if return TRUE, have read and processed any parameters, and have
 *   updated the restart point to point after the parameters.
 *   If return FALSE, was forced to suspend before reaching end of
 *   marker parameters; restart point has not been moved. Same routine
 *   will be called again after application supplies more input data.
 *
 * This approach to suspension assumes that all of a marker's parameters
 * can fit into a single input bufferload. This should hold for "normal"
 * markers. Some COM/APPn markers might have large parameter segments
 * that might not fit. If we are simply dropping such a marker, we use
 * skip_input_data to get past it, and thereby put the problem on the
 * source manager's shoulders. If we are saving the marker's contents
 * into memory, we use a slightly different convention: when forced to
 * suspend, the marker processor updates the restart point to the end of
 * what it's consumed (ie, the end of the buffer) before returning FALSE.
 * On resumption, cinfo->unread_marker still contains the marker code,
 * but the data source will point to the next chunk of marker data.
 * The marker processor must retain internal state to deal with this.
 *
 * Note that we don't bother to avoid duplicate trace messages if a
 * suspension occurs within marker parameters. Other side effects
 * require more care.
 */

```

```

LOCAL(boolean)
get_soi (j_decompress_ptr cinfo)
/* Process an SOI marker */
{
    int i;

    TRACEMS(cinfo, 1, JTRC_SOI);

    if (cinfo->marker->saw_SOI)
        ERREXIT(cinfo, JERR_SOI_DUPLICATE);

    /* Reset all parameters that are defined to be reset by SOI */

    for (i = 0; i < NUM_ARITH_TBLS; i++) {
        cinfo->arith_dc_L[i] = 0;
        cinfo->arith_dc_U[i] = 1;
        cinfo->arith_ac_K[i] = 5;
    }
    cinfo->restart_interval = 0;

    /* Set initial assumptions for colorspace etc */

    cinfo->jpeg_color_space = JCS_UNKNOWN;
    cinfo->CCIR601_sampling = FALSE; /* Assume non-CCIR sampling??? */

    cinfo->saw_JFIF_marker = FALSE;
    cinfo->JFIF_major_version = 1; /* set default JFIF APP0 values */
    cinfo->JFIF_minor_version = 1;
    cinfo->density_unit = 0;
    cinfo->X_density = 1;
    cinfo->Y_density = 1;
    cinfo->saw_Adobe_marker = FALSE;
    cinfo->Adobe_transform = 0;

    cinfo->marker->saw_SOI = TRUE;

    return TRUE;
}

```

```

LOCAL(boolean)
get_sof (j_decompress_ptr cinfo, boolean is_prog, boolean is_arith)
/* Process a SOFn marker */
{
    INT32 length;

```

```

int c, ci;
jpeg_component_info * comp;
INPUT_VARS(cinfo);

cinfo->progressive_mode = is_prog;
cinfo->arith_code = is_arith;

INPUT_2BYTES(cinfo, length, return FALSE);

INPUT_BYTE(cinfo, cinfo->data_precision, return FALSE);
INPUT_2BYTES(cinfo, cinfo->image_height, return FALSE);
INPUT_2BYTES(cinfo, cinfo->image_width, return FALSE);
INPUT_BYTE(cinfo, cinfo->num_components, return FALSE);

length -= 8;

TRACEMS4(cinfo, 1, JTRC_SOF, cinfo->unread_marker,
(int) cinfo->image_width, (int) cinfo->image_height,
cinfo->num_components);

if (cinfo->marker->saw_SOF)
    ERREXIT(cinfo, JERR_SOF_DUPLICATE);

/* We don't support files in which the image height is initially specified */
/* as 0 and is later redefined by DNL. As long as we have to check that, */
/* might as well have a general sanity check. */
if (cinfo->image_height <= 0 || cinfo->image_width <= 0
    || cinfo->num_components <= 0)
    ERREXIT(cinfo, JERR_EMPTY_IMAGE);

if (length != (cinfo->num_components * 3))
    ERREXIT(cinfo, JERR_BAD_LENGTH);

if (cinfo->comp_info == NULL) /* do only once, even if suspend */
    cinfo->comp_info = (jpeg_component_info *) (*cinfo->mem->alloc_small)
        ((j_common_ptr) cinfo, JPOOL_IMAGE,
        cinfo->num_components * sizeof(jpeg_component_info));

for (ci = 0, compptr = cinfo->comp_info; ci < cinfo->num_components;
    ci++, compptr++) {
    compptr->component_index = ci;
    INPUT_BYTE(cinfo, compptr->component_id, return FALSE);
    INPUT_BYTE(cinfo, c, return FALSE);
    compptr->h_samp_factor = (c >> 4) & 15;
    compptr->v_samp_factor = (c & 15);
    INPUT_BYTE(cinfo, compptr->quant_tbl_no, return FALSE);

    TRACEMS4(cinfo, 1, JTRC_SOF_COMPONENT,
        compptr->component_id, compptr->h_samp_factor,
        compptr->v_samp_factor, compptr->quant_tbl_no);

    cinfo->marker->saw_SOF = TRUE;

    INPUT_SYNC(cinfo);
    return TRUE;
}

LOCAL(boolean)
get_sos (j_decompress_ptr cinfo)
/* Process a SOS marker */
{
    INT32 length;
    int i, ci, n, c, cc;
    jpeg_component_info * compptr;
    INPUT_VARS(cinfo);

    if (! cinfo->marker->saw_SOF)
        ERREXIT(cinfo, JERR_SOS_NO_SOF);

    INPUT_2BYTES(cinfo, length, return FALSE);

    INPUT_BYTE(cinfo, n, return FALSE); /* Number of components */

    TRACEMS1(cinfo, 1, JTRC_SOS, n);

    if (length != (n * 2 + 6) || n < 1 || n > MAX_COMPS_IN_SCAN)
        ERREXIT(cinfo, JERR_BAD_LENGTH);
}

```

```

cinfo->comps_in_scan = n;

/* Collect the component-specific parameters */
for (i = 0; i < n; i++) {
    INPUT_BYTE(cinfo, cc, return FALSE);
    INPUT_BYTE(cinfo, c, return FALSE);

    for (ci = 0, compptr = cinfo->comp_info; ci < cinfo->num_components;
        ci++, compptr++) {
        if (cc == compptr->component_id)
            goto id_found;
    }

    ERREXIT1(cinfo, JERR_BAD_COMPONENT_ID, cc);
}

id_found:

cinfo->cur_comp_info[i] = compptr;
compptr->dc_tbl_no = (c >> 4) & 15;
compptr->ac_tbl_no = (c & 15);

TRACEMS3(cinfo, 1, JTRC_SOS_COMPONENT, cc,
    compptr->dc_tbl_no, compptr->ac_tbl_no);
}

/* Collect the additional scan parameters Ss, Se, Ah/Al. */
INPUT_BYTE(cinfo, c, return FALSE);
cinfo->Ss = c;
INPUT_BYTE(cinfo, c, return FALSE);
cinfo->Se = c;
INPUT_BYTE(cinfo, c, return FALSE);
cinfo->Ah = (c >> 4) & 15;
cinfo->Al = (c & 15);

TRACEMS4(cinfo, 1, JTRC_SOS_PARAMS, cinfo->Ss, cinfo->Se,
    cinfo->Ah, cinfo->Al);

/* Prepare to scan data & restart markers */
cinfo->marker->next_restart_num = 0;

/* Count another SOS marker */
cinfo->input_scan_number++;

= INPUT_SYNC(cinfo);
return TRUE;

#ifdef D_ARITH_CODING_SUPPORTED
LOCAL(boolean)
get_dac (j_decompress_ptr cinfo)
/* Process a DAC marker */
{
    INT32 length;
    int index, val;
    INPUT_VARS(cinfo);

    INPUT_2BYTES(cinfo, length, return FALSE);
    length -= 2;

    while (length > 0) {
        INPUT_BYTE(cinfo, index, return FALSE);
        INPUT_BYTE(cinfo, val, return FALSE);

        length -= 2;

        TRACEMS2(cinfo, 1, JTRC_DAC, index, val);

        if (index < 0 || index >= (2*NUM_ARITH_TBLS))
            ERREXIT1(cinfo, JERR_DAC_INDEX, index);

        if (index >= NUM_ARITH_TBLS) { /* define AC table */
            cinfo->arith_ac_K[index-NUM_ARITH_TBLS] = (UINT8) val;
        } else { /* define DC table */
            cinfo->arith_dc_L[index] = (UINT8) (val & 0x0F);
            cinfo->arith_dc_U[index] = (UINT8) (val >> 4);
            if (cinfo->arith_dc_L[index] > cinfo->arith_dc_U[index])
                ERREXIT1(cinfo, JERR_DAC_VALUE, val);
        }
    }
}

```

```

    }
}

if (length != 0)
    ERREXIT(cinfo, JERR_BAD_LENGTH);

INPUT_SYNC(cinfo);
return TRUE;
}

#else /* ! D_ARITH_CODING_SUPPORTED */

#define get_dac(cinfo) skip_variable(cinfo)

#endif /* D_ARITH_CODING_SUPPORTED */

LOCAL(boolean)
get_dht (j_decompress_ptr cinfo)
/* Process a DHT marker */
{
    INT32 length;
    UINT8 bits[17];
    UINT8 huffval[256];
    int i, index, count;
    JHUFF_TBL **htblptr;
    INPUT_VARS(cinfo);

    INPUT_2BYTES(cinfo, length, return FALSE);
    length -= 2;

    while (length > 16) {
        INPUT_BYTE(cinfo, index, return FALSE);

        TRACEMS1(cinfo, 1, JTRC_DHT, index);

        bits[0] = 0;
        count = 0;
        for (i = 1; i <= 16; i++) {
            INPUT_BYTE(cinfo, bits[i], return FALSE);
            count += bits[i];
        }

        length -= 1 + 16;

        TRACEMS8(cinfo, 2, JTRC_HUFFBITS,
            bits[1], bits[2], bits[3], bits[4],
            bits[5], bits[6], bits[7], bits[8]);
        TRACEMS8(cinfo, 2, JTRC_HUFFBITS,
            bits[9], bits[10], bits[11], bits[12],
            bits[13], bits[14], bits[15], bits[16]);

        /* Here we just do minimal validation of the counts to avoid walking
         * off the end of our table space.  jdhuft.c will check more carefully.
         */
        if (count > 256 || ((INT32) count) > length)
            ERREXIT(cinfo, JERR_BAD_HUFF_TABLE);

        for (i = 0; i < count; i++)
            INPUT_BYTE(cinfo, huffval[i], return FALSE);

        length -= count;

        if (index & 0x10) { /* AC table definition */
            index -= 0x10;
            htblptr = &cinfo->ac_huff_tbl_ptrs[index];
        } else { /* DC table definition */
            htblptr = &cinfo->dc_huff_tbl_ptrs[index];
        }

        if (index < 0 || index >= NUM_HUFF_TBLS)
            ERREXIT1(cinfo, JERR_DHT_INDEX, index);

        if (*htblptr == NULL)
            *htblptr = jpeg_alloc_huff_table((j_common_ptr) cinfo);

        MEMCOPY((*htblptr)->bits, bits, SIZEOF((*htblptr)->bits));
        MEMCOPY((*htblptr)->huffval, huffval, SIZEOF((*htblptr)->huffval));
    }
}

```

```

    if (length != 0)
        ERREXIT(cinfo, JERR_BAD_LENGTH);

    INPUT_SYNC(cinfo);
    return TRUE;
}

LOCAL(boolean)
get_dqt (j_decompress_ptr cinfo)
/* Process a DQT marker */
{
    INT32 length;
    int n, i, prec;
    unsigned int tmp;
    JQUANT_TBL *quant_ptr;
    INPUT_VARS(cinfo);

    INPUT_2BYTES(cinfo, length, return FALSE);
    length -= 2;

    while (length > 0) {
        INPUT_BYTE(cinfo, n, return FALSE);
        prec = n >> 4;
        n &= 0x0F;

        TRACEMS2(cinfo, 1, JTRC_DQT, n, prec);

        if (n >= NUM_QUANT_TBLS)
            ERREXIT1(cinfo, JERR_DQT_INDEX, n);

        if (cinfo->quant_tbl_ptrs[n] == NULL)
            cinfo->quant_tbl_ptrs[n] = jpeg_alloc_quant_table((j_common_ptr) cinfo);
        quant_ptr = cinfo->quant_tbl_ptrs[n];

        for (i = 0; i < DCTSIZE2; i++) {
            if (prec)
                INPUT_2BYTES(cinfo, tmp, return FALSE);
            else
                INPUT_BYTE(cinfo, tmp, return FALSE);
            /* We convert the zigzag-order table to natural array order. */
            quant_ptr->quantval[jpeg_natural_order[i]] = (UINT16) tmp;
        }

        if (cinfo->err->trace_level >= 2) {
            for (i = 0; i < DCTSIZE2; i += 8) {
                TRACEMS8(cinfo, 2, JTRC_QUANTVALS,
                    quant_ptr->quantval[i], quant_ptr->quantval[i+1],
                    quant_ptr->quantval[i+2], quant_ptr->quantval[i+3],
                    quant_ptr->quantval[i+4], quant_ptr->quantval[i+5],
                    quant_ptr->quantval[i+6], quant_ptr->quantval[i+7]);
            }
        }

        length -= DCTSIZE2+1;
        if (prec) length -= DCTSIZE2;
    }

    if (length != 0)
        ERREXIT(cinfo, JERR_BAD_LENGTH);

    INPUT_SYNC(cinfo);
    return TRUE;
}

LOCAL(boolean)
get_dri (j_decompress_ptr cinfo)
/* Process a DRI marker */
{
    INT32 length;
    unsigned int tmp;
    INPUT_VARS(cinfo);

    INPUT_2BYTES(cinfo, length, return FALSE);

    if (length != 4)
        ERREXIT(cinfo, JERR_BAD_LENGTH);

    INPUT_2BYTES(cinfo, tmp, return FALSE);

```

```

TRACEMS1(cinfo, 1, JTRC_DRImp);

cinfo->restart_interval = tmp;

INPUT_SYNC(cinfo);
return TRUE;
}

/*
 * Routines for processing APPn and COM markers.
 * These are either saved in memory or discarded, per application request.
 * APP0 and APP14 are specially checked to see if they are
 * JFIF and Adobe markers, respectively.
 */

#define APP0_DATA_LEN 14 /* Length of interesting data in APP0 */
#define APP14_DATA_LEN 12 /* Length of interesting data in APP14 */
#define APPN_DATA_LEN 14 /* Must be the largest of the above!! */

LOCAL(void)
examine_app0 (j_decompress_ptr cinfo, JOCTET * data,
              unsigned int datalen, INT32 remaining)
/* Examine first few bytes from an APP0.
 * Take appropriate action if it is a JFIF marker.
 * datalen is # of bytes at data[], remaining is length of rest of marker data.
 */
{
    INT32 totallen = (INT32) datalen + remaining;

    if (datalen >= APP0_DATA_LEN &&
        GETJOCTET(data[0]) == 0x4A &&
        GETJOCTET(data[1]) == 0x46 &&
        GETJOCTET(data[2]) == 0x49 &&
        GETJOCTET(data[3]) == 0x46 &&
        GETJOCTET(data[4]) == 0) {
        /* Found JFIF APP0 marker: save info */
        cinfo->saw_JFIF_marker = TRUE;
        cinfo->JFIF_major_version = GETJOCTET(data[5]);
        cinfo->JFIF_minor_version = GETJOCTET(data[6]);
        cinfo->density_unit = GETJOCTET(data[7]);
        cinfo->X_density = (GETJOCTET(data[8]) << 8) + GETJOCTET(data[9]);
        cinfo->Y_density = (GETJOCTET(data[10]) << 8) + GETJOCTET(data[11]);
        /* Check version.
         * Major version must be 1, anything else signals an incompatible change.
         * (We used to treat this as an error, but now it's a nonfatal warning,
         * because some bozo at Hijaak couldn't read the spec.)
         * Minor version should be 0..2, but process anyway if newer.
         */
        if (cinfo->JFIF_major_version != 1)
            WARNMS2(cinfo, JWRN_JFIF_MAJOR,
                   cinfo->JFIF_major_version, cinfo->JFIF_minor_version);
        /* Generate trace messages */
        TRACEMS5(cinfo, 1, JTRC_JFIF,
                 cinfo->JFIF_major_version, cinfo->JFIF_minor_version,
                 cinfo->X_density, cinfo->Y_density, cinfo->density_unit);
        /* Validate thumbnail dimensions and issue appropriate messages */
        if (GETJOCTET(data[12]) | GETJOCTET(data[13]))
            TRACEMS2(cinfo, 1, JTRC_JFIF_THUMBNAIL,
                     GETJOCTET(data[12]), GETJOCTET(data[13]));
        totallen -= APP0_DATA_LEN;
        if (totallen !=
            ((INT32)GETJOCTET(data[12]) * (INT32)GETJOCTET(data[13]) * (INT32) 3))
            TRACEMS1(cinfo, 1, JTRC_JFIF_BADTHUMBNAILSIZE, (int) totallen);
    } else if (datalen >= 6 &&
        GETJOCTET(data[0]) == 0x4A &&
        GETJOCTET(data[1]) == 0x46 &&
        GETJOCTET(data[2]) == 0x58 &&
        GETJOCTET(data[3]) == 0x58 &&
        GETJOCTET(data[4]) == 0) {
        /* Found JFIF "JFXX" extension APP0 marker */
        /* The library doesn't actually do anything with these,
         * but we try to produce a helpful trace message.
         */
        switch (GETJOCTET(data[5])) {
        case 0x10:
            TRACEMS1(cinfo, 1, JTRC_THUMB_JPEG, (int) totallen);
            break;

```

```

case 0x11:
    TRACEMS1(cinfo, 1, JTRC_THUMB_PALETTE, (int) totalen);
    break;
case 0x13:
    TRACEMS1(cinfo, 1, JTRC_THUMB_RGB, (int) totalen);
    break;
default:
    TRACEMS2(cinfo, 1, JTRC_JFIF_EXTENSION,
        GETJOCTET(data[5]), (int) totalen);
    break;
}
} else {
    /* Start of APP0 does not match "JFIF" or "JFXX", or too short */
    TRACEMS1(cinfo, 1, JTRC_APP0, (int) totalen);
}
}

LOCAL(void)
examine_app14 (j_decompress_ptr cinfo, JOCTET * data,
    unsigned int datalen, INT32 remaining)
/* Examine first few bytes from an APP14.
 * Take appropriate action if it is an Adobe marker.
 * datalen is # of bytes at data[], remaining is length of rest of marker data.
 */
{
    unsigned int version, flags0, flags1, transform;

    if (datalen >= APP14_DATA_LEN &&
        GETJOCTET(data[0]) == 0x41 &&
        GETJOCTET(data[1]) == 0x64 &&
        GETJOCTET(data[2]) == 0x6F &&
        GETJOCTET(data[3]) == 0x62 &&
        GETJOCTET(data[4]) == 0x65) {
        /* Found Adobe APP14 marker */
        version = (GETJOCTET(data[5]) << 8) + GETJOCTET(data[6]);
        flags0 = (GETJOCTET(data[7]) << 8) + GETJOCTET(data[8]);
        flags1 = (GETJOCTET(data[9]) << 8) + GETJOCTET(data[10]);
        transform = GETJOCTET(data[11]);
        TRACEMS4(cinfo, 1, JTRC_ADOBE, version, flags0, flags1, transform);
        cinfo->saw_Adobe_marker = TRUE;
        cinfo->Adobe_transform = (UINT8) transform;
    } else {
        /* Start of APP14 does not match "Adobe", or too short */
        TRACEMS1(cinfo, 1, JTRC_APP14, (int) (datalen + remaining));
    }
}

METHODDEF(boolean)
get_interesting_appn (j_decompress_ptr cinfo)
/* Process an APP0 or APP14 marker without saving it */
{
    INT32 length;
    JOCTET b[APPN_DATA_LEN];
    unsigned int i, numtoread;
    INPUT_VARS(cinfo);

    INPUT_2BYTES(cinfo, length, return FALSE);
    length -= 2;

    /* get the interesting part of the marker data */
    if (length >= APPN_DATA_LEN)
        numtoread = APPN_DATA_LEN;
    else if (length > 0)
        numtoread = (unsigned int) length;
    else
        numtoread = 0;
    for (i = 0; i < numtoread; i++)
        INPUT_BYTE(cinfo, b[i], return FALSE);
    length -= numtoread;

    /* process it */
    switch (cinfo->unread_marker) {
    case M_APP0:
        examine_app0(cinfo, (JOCTET *) b, numtoread, length);
        break;
    case M_APP14:
        examine_app14(cinfo, (JOCTET *) b, numtoread, length);
        break;
    }
}

```



09697592 402700

```

default:
/* can't get here unless save_markers chooses wrong proc */
ERREXIT1(cinfo, JERR_UNKNOWN_MARKER, cinfo->unread_marker);
break;
)

/* skip any remaining data -- could be lots */
INPUT_SYNC(cinfo);
if (length > 0)
    (*cinfo->src->skip_input_data) (cinfo, (long) length);

return TRUE;
}

#ifdef SAVE_MARKERS_SUPPORTED

METHODDEF(boolean)
save_marker (j_decompress_ptr cinfo)
/* Save an APPn or COM marker into the marker list */
{
    my_marker_ptr marker = (my_marker_ptr) cinfo->marker;
    jpeg_saved_marker_ptr cur_marker = marker->cur_marker;
    unsigned int bytes_read, data_length;
    JOCTET * data;
    INT32 length = 0;
    INPUT_VARS(cinfo);

    if (cur_marker == NULL) {
        /* begin reading a marker */
        INPUT_2BYTES(cinfo, length, return FALSE);
        length -= 2;
        if (length >= 0) { /* watch out for bogus length word */
            /* figure out how much we want to save */
            unsigned int limit;
            if (cinfo->unread_marker == (int) M_COM)
                limit = marker->length_limit_COM;
            else
                limit = marker->length_limit_APPn[cinfo->unread_marker - (int) M_APP0];
            if ((unsigned int) length < limit)
                limit = (unsigned int) length;
            /* allocate and initialize the marker item */
            cur_marker = (jpeg_saved_marker_ptr)
                (*cinfo->mem->alloc_large) ((j_common_ptr) cinfo, JPOOL_IMAGE,
                    sizeof(struct jpeg_marker_struct) + limit);
            cur_marker->next = NULL;
            cur_marker->marker = (UINT8) cinfo->unread_marker;
            cur_marker->original_length = (unsigned int) length;
            cur_marker->data_length = limit;
            /* data area is just beyond the jpeg_marker_struct */
            data = cur_marker->data = (JOCTET *) (cur_marker + 1);
            marker->cur_marker = cur_marker;
            marker->bytes_read = 0;
            bytes_read = 0;
            data_length = limit;
        } else {
            /* deal with bogus length word */
            bytes_read = data_length = 0;
            data = NULL;
        }
    } else {
        /* resume reading a marker */
        bytes_read = marker->bytes_read;
        data_length = cur_marker->data_length;
        data = cur_marker->data + bytes_read;
    }

    while (bytes_read < data_length) {
        INPUT_SYNC(cinfo); /* move the restart point to here */
        marker->bytes_read = bytes_read;
        /* If there's not at least one byte in buffer, suspend */
        MAKE_BYTE_AVAIL(cinfo, return FALSE);
        /* Copy bytes with reasonable rapidity */
        while (bytes_read < data_length && bytes_in_buffer > 0) {
            *data++ = *next_input_byte++;
            bytes_in_buffer--;
            bytes_read++;
        }
    }
}

```

```

/* Done reading what we want to read */
if (cur_marker != NULL) { /* Will be NULL if bogus length word */
/* Add new marker to end of list */
if (cinfo->marker_list == NULL) {
cinfo->marker_list = cur_marker;
} else {
jpeg_saved_marker_ptr prev = cinfo->marker_list;
while (prev->next != NULL)
prev = prev->next;
prev->next = cur_marker;
}
/* Reset pointer & calc remaining data length */
data = cur_marker->data;
length = cur_marker->original_length - data_length;
}
/* Reset to initial state for next marker */
marker->cur_marker = NULL;

/* Process the marker if interesting; else just make a generic trace msg */
switch (cinfo->unread_marker) {
case M_APP0:
examine_app0(cinfo, data, data_length, length);
break;
case M_APP14:
examine_app14(cinfo, data, data_length, length);
break;
default:
TRACEMS2(cinfo, 1, JTRC_MISC_MARKER, cinfo->unread_marker,
(int) (data_length + length));
break;
}
}
/* skip any remaining data -- could be lots */
INPUT_SYNC(cinfo); /* do before skip_input_data */
if (length > 0)
(*cinfo->src->skip_input_data) (cinfo, (long) length);
return TRUE;
}
#endif /* SAVE_MARKERS_SUPPORTED */

METHODDEF(boolean)
skip_variable (j_decompress_ptr cinfo)
/* Skip over an unknown or uninteresting variable-length marker */
{
INT32 length;
INPUT_VARS(cinfo);
INPUT_2BYTES(cinfo, length, return FALSE);
length -= 2;
TRACEMS2(cinfo, 1, JTRC_MISC_MARKER, cinfo->unread_marker, (int) length);
INPUT_SYNC(cinfo); /* do before skip_input_data */
if (length > 0)
(*cinfo->src->skip_input_data) (cinfo, (long) length);
return TRUE;
}

/*
* Find the next JPEG marker, save it in cinfo->unread_marker.
* Returns FALSE if had to suspend before reaching a marker;
* in that case cinfo->unread_marker is unchanged.
*
* Note that the result might not be a valid marker code,
* but it will never be 0 or FF.
*/
LOCAL(boolean)
next_marker (j_decompress_ptr cinfo)
{
int c;
INPUT_VARS(cinfo);
for (;;) {
INPUT_BYTE(cinfo, c, return FALSE);

```

```

/* Skip any non-FF bytes.
 * This may look a bit inefficient, but it will not occur in a valid file.
 * We sync after each discarded byte so that a suspending data source
 * can discard the byte from its buffer.
 */
while (c != 0xFF) {
    cinfo->marker->discarded_bytes++;
    INPUT_SYNC(cinfo);
    INPUT_BYTE(cinfo, c, return FALSE);
}
/* This loop swallows any duplicate FF bytes. Extra FFs are legal as
 * pad bytes, so don't count them in discarded_bytes. We assume there
 * will not be so many consecutive FF bytes as to overflow a suspending
 * data source's input buffer.
 */
do {
    INPUT_BYTE(cinfo, c, return FALSE);
} while (c == 0xFF);
if (c != 0)
    break; /* found a valid marker, exit loop */
/* Reach here if we found a stuffed-zero data sequence (FF/00).
 * Discard it and loop back to try again.
 */
cinfo->marker->discarded_bytes += 2;
INPUT_SYNC(cinfo);
}

if (cinfo->marker->discarded_bytes != 0) {
    WARNMS2(cinfo, JWRN_EXTRANEEOUS_DATA, cinfo->marker->discarded_bytes, c);
    cinfo->marker->discarded_bytes = 0;
}

cinfo->unread_marker = c;
INPUT_SYNC(cinfo);
return TRUE;
}

LOCAL(boolean)
first_marker(j_decompress_ptr cinfo)
/* Like next_marker, but used to obtain the initial SOI marker. */
/* For this marker, we do not allow preceding garbage or fill; otherwise,
 * we might well scan an entire input file before realizing it ain't JPEG.
 * If an application wants to process non-JFIF files, it must seek to the
 * SOI before calling the JPEG library.
 */
{
    int c, c2;
    INPUT_VARS(cinfo);

    INPUT_BYTE(cinfo, c, return FALSE);
    INPUT_BYTE(cinfo, c2, return FALSE);
    if (c != 0xFF || c2 != (int) M_SOI)
        ERREXIT2(cinfo, JERR_NO_SOI, c, c2);

    cinfo->unread_marker = c2;

    INPUT_SYNC(cinfo);
    return TRUE;
}

/*
 * Read markers until SOS or EOI.
 *
 * Returns same codes as are defined for jpeg_consume_input:
 * JPEG_SUSPENDED, JPEG_REACHED_SOS, or JPEG_REACHED_EOI.
 */

METHODDEF(int)
read_markers(j_decompress_ptr cinfo)
{
    /* Outer loop repeats once for each marker. */
    for (;;) {
        /* Collect the marker proper, unless we already did. */
        /* NB: first_marker() enforces the requirement that SOI appear first. */
        if (cinfo->unread_marker == 0) {
            if (!cinfo->marker->saw_SOI) {
                if (!first_marker(cinfo))

```

```

    return JPEG_SUSPENDED;
} else {
if (! next_marker(cinfo))
    return JPEG_SUSPENDED;
}
}
/* At this point cinfo->unread_marker contains the marker code and the
 * input point is just past the marker proper, but before any parameters.
 * A suspension will cause us to return with this state still true.
 */
switch (cinfo->unread_marker) {
case M_SOI:
    if (! get_soi(cinfo))
        return JPEG_SUSPENDED;
    break;

case M_SOF0:          /* Baseline */
case M_SOF1:          /* Extended sequential, Huffman */
    if (! get_sof(cinfo, FALSE, FALSE))
        return JPEG_SUSPENDED;
    break;

case M_SOF2:          /* Progressive, Huffman */
    if (! get_sof(cinfo, TRUE, FALSE))
        return JPEG_SUSPENDED;
    break;

case M_SOF9:          /* Extended sequential, arithmetic */
    if (! get_sof(cinfo, FALSE, TRUE))
        return JPEG_SUSPENDED;
    break;

case M_SOF10:         /* Progressive, arithmetic */
    if (! get_sof(cinfo, TRUE, TRUE))
        return JPEG_SUSPENDED;
    break;

/* Currently unsupported SOFn types */
case M_SOF3:          /* Lossless, Huffman */
case M_SOF5:          /* Differential sequential, Huffman */
case M_SOF6:          /* Differential progressive, Huffman */
case M_SOF7:          /* Differential lossless, Huffman */
case M_JPG:           /* Reserved for JPEG extensions */
case M_SOF11:         /* Lossless, arithmetic */
case M_SOF13:         /* Differential sequential, arithmetic */
case M_SOF14:         /* Differential progressive, arithmetic */
case M_SOF15:         /* Differential lossless, arithmetic */
    ERREXIT1(cinfo, JERR_SOF_UNSUPPORTED, cinfo->unread_marker);
    break;

case M_SOS:
    if (! get_sos(cinfo))
        return JPEG_SUSPENDED;
    cinfo->unread_marker = 0; /* processed the marker */
    return JPEG_REACHED_SOS;

case M_EOI:
    TRACEMS(cinfo, 1, JTRC_EOI);
    cinfo->unread_marker = 0; /* processed the marker */
    return JPEG_REACHED_EOI;

case M_DAC:
    if (! get_dac(cinfo))
        return JPEG_SUSPENDED;
    break;

case M_DHT:
    if (! get_dht(cinfo))
        return JPEG_SUSPENDED;
    break;

case M_DQT:
    if (! get_dqt(cinfo))
        return JPEG_SUSPENDED;
    break;

case M_DRI:
    if (! get_dri(cinfo))
        return JPEG_SUSPENDED;
    break;

```

```

case M_APP0:
case M_APP1:
case M_APP2:
case M_APP3:
case M_APP4:
case M_APP5:
case M_APP6:
case M_APP7:
case M_APP8:
case M_APP9:
case M_APP10:
case M_APP11:
case M_APP12:
case M_APP13:
case M_APP14:
case M_APP15:
    if (! ((my_marker_ptr) cinfo->marker)->process_APPn[
        cinfo->unread_marker - (int) M_APP0]) (cinfo))
return JPEG_SUSPENDED;
    break;

case M_COM:
    if (! ((my_marker_ptr) cinfo->marker)->process_COM) (cinfo))
return JPEG_SUSPENDED;
    break;

case M_RST0:          /* these are all parameterless */
case M_RST1:
case M_RST2:
case M_RST3:
case M_RST4:
case M_RST5:
case M_RST6:
case M_RST7:
case M_TEM:
    TRACEMS1(cinfo, 1, JTRC_PARMLESS_MARKER, cinfo->unread_marker);
    break;

case M_DNL:           /* Ignore DNL ... perhaps the wrong thing */
    if (! skip_variable(cinfo))
return JPEG_SUSPENDED;
    break;

default:              /* must be DHP, EXP, JPGn, or RESn */
    /* For now, we treat the reserved markers as fatal errors since they are
     * likely to be used to signal incompatible JPEG Part 3 extensions.
     * Once the JPEG 3 version-number marker is well defined, this code
     * ought to change!
     */
    ERREXIT1(cinfo, JERR_UNKNOWN_MARKER, cinfo->unread_marker);
    break;
}
/* Successfully processed marker, so reset state variable */
cinfo->unread_marker = 0;
} /* end loop */
}

/*
 * Read a restart marker, which is expected to appear next in the datastream;
 * if the marker is not there, take appropriate recovery action.
 * Returns FALSE if suspension is required.
 *
 * This is called by the entropy decoder after it has read an appropriate
 * number of MCUs. cinfo->unread_marker may be nonzero if the entropy decoder
 * has already read a marker from the data source. Under normal conditions
 * cinfo->unread_marker will be reset to 0 before returning; if not reset,
 * it holds a marker which the decoder will be unable to read past.
 */

METHODDEF(boolean)
read_restart_marker (j_decompress_ptr cinfo)
{
    /* Obtain a marker unless we already did. */
    /* Note that next_marker will complain if it skips any data. */
    if (cinfo->unread_marker == 0) {
        if (! next_marker(cinfo))
            return FALSE;
    }
}

```

```

if (cinfo->unread_marker ==
    ((int) M_RST0 + cinfo->marker->next_restart_num)) {
    /* Normal case --- swallow the marker and let entropy decoder continue */
    TRACEMS1(cinfo, 3, JTRC_RST, cinfo->marker->next_restart_num);
    cinfo->unread_marker = 0;
} else {
    /* Uh-oh, the restart markers have been messed up. */
    /* Let the data source manager determine how to resync. */
    if (! (*cinfo->src->resync_to_restart) (cinfo,
        cinfo->marker->next_restart_num))
        return FALSE;
}

/* Update next-restart state */
cinfo->marker->next_restart_num = (cinfo->marker->next_restart_num + 1) & 7;

return TRUE;
}

```

```

/*
 * This is the default resync_to_restart method for data source managers
 * to use if they don't have any better approach. Some data source managers
 * may be able to back up, or may have additional knowledge about the data
 * which permits a more intelligent recovery strategy; such managers would
 * presumably supply their own resync method.
 *
 * read_restart_marker calls resync_to_restart if it finds a marker other than
 * the restart marker it was expecting. (This code is *not* used unless
 * a nonzero restart interval has been declared.) cinfo->unread_marker is
 * the marker code actually found (might be anything, except 0 or FF).
 * The desired restart marker number (0..7) is passed as a parameter.
 * This routine is supposed to apply whatever error recovery strategy seems
 * appropriate in order to position the input stream to the next data segment.
 * Note that cinfo->unread_marker is treated as a marker appearing before
 * the current data-source input point; usually it should be reset to zero
 * before returning.
 * Returns FALSE if suspension is required.
 *
 * This implementation is substantially constrained by wanting to treat the
 * input as a data stream; this means we can't back up. Therefore, we have
 * only the following actions to work with:
 *
 * 1. Simply discard the marker and let the entropy decoder resume at next
 *    byte of file.
 *
 * 2. Read forward until we find another marker, discarding intervening
 *    data. (In theory we could look ahead within the current bufferload,
 *    without having to discard data if we don't find the desired marker.
 *    This idea is not implemented here, in part because it makes behavior
 *    dependent on buffer size and chance buffer-boundary positions.)
 *
 * 3. Leave the marker unread (by failing to zero cinfo->unread_marker).
 *    This will cause the entropy decoder to process an empty data segment,
 *    inserting dummy zeroes, and then we will reprocess the marker.
 *
 * #2 is appropriate if we think the desired marker lies ahead, while #3 is
 * appropriate if the found marker is a future restart marker (indicating
 * that we have missed the desired restart marker, probably because it got
 * corrupted).
 * We apply #2 or #3 if the found marker is a restart marker no more than
 * two counts behind or ahead of the expected one. We also apply #2 if the
 * found marker is not a legal JPEG marker code (it's certainly bogus data).
 * If the found marker is a restart marker more than 2 counts away, we do #1
 * (too much risk that the marker is erroneous; with luck we will be able to
 * resync at some future point).
 * For any valid non-restart JPEG marker, we apply #3. This keeps us from
 * overrunning the end of a scan. An implementation limited to single-scan
 * files might find it better to apply #2 for markers other than EOI, since
 * any other marker would have to be bogus data in that case.
 */

```

```

GLOBAL(boolean)
jpeg_resync_to_restart (j_decompress_ptr cinfo, int desired)
{
    int marker = cinfo->unread_marker;
    int action = 1;

    /* Always put up a warning. */
    WARNMS2(cinfo, JWRN_MUST_RESYNC, marker, desired);

    /* Outer loop handles repeated decision after scanning forward. */

```

```

for (;;) {
    if (marker < (int) M_SOF)
        action = 2; /* invalid marker */
    else if (marker < (int) M_RST0 || marker > (int) M_RST7)
        action = 3; /* valid non-restart marker */
    else {
        if (marker == ((int) M_RST0 + ((desired+1) & 7)) ||
            marker == ((int) M_RST0 + ((desired+2) & 7)))
            action = 3; /* one of the next two expected restarts */
        else if (marker == ((int) M_RST0 + ((desired-1) & 7)) ||
            marker == ((int) M_RST0 + ((desired-2) & 7)))
            action = 2; /* a prior restart, so advance */
        else
            action = 1; /* desired restart or too far away */
    }
    TRACE2(cinfo, 4, JTRC_RECOVERY_ACTION, marker, action);
    switch (action) {
    case 1:
        /* Discard marker and let entropy decoder resume processing. */
        cinfo->unread_marker = 0;
        return TRUE;
    case 2:
        /* Scan to the next marker, and repeat the decision loop. */
        if (! next_marker(cinfo))
            return FALSE;
        marker = cinfo->unread_marker;
        break;
    case 3:
        /* Return without advancing past this marker. */
        /* Entropy decoder will be forced to process an empty segment. */
        return TRUE;
    }
} /* end loop */

/* Reset marker processing state to begin a fresh datastream. */

METHODDEF(void)
reset_marker_reader (j_decompress_ptr cinfo)
{
    my_marker_ptr marker = (my_marker_ptr) cinfo->marker;

    cinfo->comp_info = NULL; /* until allocated by get_sof */
    cinfo->input_scan_number = 0; /* no SOS seen yet */
    cinfo->unread_marker = 0; /* no pending marker */
    marker->pub.saw_SOI = FALSE; /* set internal state too */
    marker->pub.saw_SOF = FALSE;
    marker->pub.discarded_bytes = 0;
    marker->cur_marker = NULL;
}

/*
 * Initialize the marker reader module.
 * This is called only once, when the decompression object is created.
 */

GLOBAL(void)
jinit_marker_reader (j_decompress_ptr cinfo)
{
    my_marker_ptr marker;
    int i;

    /* Create subobject in permanent pool */
    marker = (my_marker_ptr)
        (*cinfo->mem->alloc_small) ((j_common_ptr) cinfo, JPOOL_PERMANENT,
            sizeof(my_marker_reader));
    cinfo->marker = (struct jpeg_marker_reader *) marker;
    /* Initialize public method pointers */
    marker->pub.reset_marker_reader = reset_marker_reader;
    marker->pub.read_markers = read_markers;
    marker->pub.read_restart_marker = read_restart_marker;
    /* Initialize COM/APPn processing.
     * By default, we examine and then discard APP0 and APP14,
     * but simply discard COM and all other APPn.
     */
    marker->process_COM = skip_variable;

```



```

marker->length_limit_COM = 0;
for (i = 0; i < 16; i++) {
    marker->process_APPn[i] = skip_variable;
    marker->length_limit_APPn[i] = 0;
}
marker->process_APPn[0] = get_interesting_appn;
marker->process_APPn[14] = get_interesting_appn;
/* Reset marker processing state */
reset_marker_reader(cinfo);
}

/*
 * Control saving of COM and APPn markers into marker_list.
 */

#ifdef SAVE_MARKERS_SUPPORTED

GLOBAL(void)
jpeg_save_markers (j_decompress_ptr cinfo, int marker_code,
    unsigned int length_limit)
{
    my_marker_ptr marker = (my_marker_ptr) cinfo->marker;
    long maxlength;
    jpeg_marker_parser_method processor;

    /* Length limit mustn't be larger than what we can allocate
     * (should only be a concern in a 16-bit environment).
     */
    maxlength = cinfo->mem->max_alloc_chunk - SIZEOF(struct jpeg_marker_struct);
    if (((long) length_limit) > maxlength)
        length_limit = (unsigned int) maxlength;

    /* Choose processor routine to use.
     * APP0/APP14 have special requirements.
     */
    if (length_limit) {
        processor = save_marker;
        /* If saving APP0/APP14, save at least enough for our internal use. */
        if (marker_code == (int) M_APP0 && length_limit < APP0_DATA_LEN)
            length_limit = APP0_DATA_LEN;
        else if (marker_code == (int) M_APP14 && length_limit < APP14_DATA_LEN)
            length_limit = APP14_DATA_LEN;
    } else {
        processor = skip_variable;
        /* If discarding APP0/APP14, use our regular on-the-fly processor. */
        if (marker_code == (int) M_APP0 || marker_code == (int) M_APP14)
            processor = get_interesting_appn;
    }

    if (marker_code == (int) M_COM) {
        marker->process_COM = processor;
        marker->length_limit_COM = length_limit;
    } else if (marker_code >= (int) M_APP0 && marker_code <= (int) M_APP15) {
        marker->process_APPn[marker_code - (int) M_APP0] = processor;
        marker->length_limit_APPn[marker_code - (int) M_APP0] = length_limit;
    } else
        ERREXIT1(cinfo, JERR_UNKNOWN_MARKER, marker_code);
}

#endif /* SAVE_MARKERS_SUPPORTED */

/*
 * Install a special processing method for COM or APPn markers.
 */

GLOBAL(void)
jpeg_set_marker_processor (j_decompress_ptr cinfo, int marker_code,
    jpeg_marker_parser_method routine)
{
    my_marker_ptr marker = (my_marker_ptr) cinfo->marker;

    if (marker_code == (int) M_COM)
        marker->process_COM = routine;
    else if (marker_code >= (int) M_APP0 && marker_code <= (int) M_APP15)
        marker->process_APPn[marker_code - (int) M_APP0] = routine;
    else
        ERREXIT1(cinfo, JERR_UNKNOWN_MARKER, marker_code);
}

```

一、  
 二、  
 三、  
 四、  
 五、  
 六、  
 七、  
 八、  
 九、  
 十、  
 十一、  
 十二、  
 十三、  
 十四、  
 十五、  
 十六、  
 十七、  
 十八、  
 十九、  
 二十、

```

/*
 * jdmaster.c
 *
 * Copyright (C) 1991-1997, Thomas G. Lane.
 * This file is part of the Independent JPEG Group's software.
 * For conditions of distribution and use, see the accompanying README file.
 *
 * This file contains master control logic for the JPEG decompressor.
 * These routines are concerned with selecting the modules to be executed
 * and with determining the number of passes and the work to be done in each
 * pass.
 */

#define JPEG_INTERNALS
#include "jinclude.h"
#include "jpeglib.h"

/* Private state */

typedef struct {
  struct jpeg_decomp_master pub; /* public fields */

  int pass_number; /* # of passes completed */

  boolean using_merged_upsample; /* TRUE if using merged upsample/cconvert */

  /* Saved references to initialized quantizer modules,
   * in case we need to switch modes.
   */
  struct jpeg_color_quantizer * quantizer_1pass;
  struct jpeg_color_quantizer * quantizer_2pass;
} my_decomp_master;

typedef my_decomp_master * my_master_ptr;

/* Determine whether merged upsample/color conversion should be used.
 * CRUCIAL: this must match the actual capabilities of jdmerge.c!
 */
LOCAL(boolean)
use_merged_upsample (j_decompress_ptr cinfo)
{
#ifdef UPSAMPLE_MERGING_SUPPORTED
  /* Merging is the equivalent of plain box-filter upsampling */
  if (cinfo->do_fancy_upsampling || cinfo->CCIR601_sampling)
    return FALSE;
  /* jdmerge.c only supports YCC=>RGB color conversion */
  if (cinfo->jpeg_color_space != JCS_YCbCr || cinfo->num_components != 3 ||
      cinfo->out_color_space != JCS_RGB ||
      cinfo->out_color_components != RGB_PIXELSIZE)
    return FALSE;
  /* and it only handles 2h1v or 2h2v sampling ratios */
  if (cinfo->comp_info[0].h_samp_factor != 2 ||
      cinfo->comp_info[1].h_samp_factor != 1 ||
      cinfo->comp_info[2].h_samp_factor != 1 ||
      cinfo->comp_info[0].v_samp_factor > 2 ||
      cinfo->comp_info[1].v_samp_factor != 1 ||
      cinfo->comp_info[2].v_samp_factor != 1)
    return FALSE;
  /* furthermore, it doesn't work if we've scaled the IDCTs differently */
  if (cinfo->comp_info[0].DCT_scaled_size != cinfo->min_DCT_scaled_size ||
      cinfo->comp_info[1].DCT_scaled_size != cinfo->min_DCT_scaled_size ||
      cinfo->comp_info[2].DCT_scaled_size != cinfo->min_DCT_scaled_size)
    return FALSE;
  /* ??? also need to test for upsample-time rescaling, when & if supported */
  return TRUE; /* by golly, it'll work... */
#else
  return FALSE;
#endif
}

/*
 * Compute output image dimensions and related values.
 * NOTE: this is exported for possible use by application.
 * Hence it mustn't do anything that can't be done twice.
 * Also note that it may be called before the master module is initialized!
 */

```

```

*/
GLOBAL(void)
jpeg_calc_output_dimensions (j_decompress_ptr cinfo)
/* Do computations that are needed before master selection phase */
{
#ifdef IDCT_SCALING_SUPPORTED
    int ci;
    jpeg_component_info *comp_ptr;
#endif

    /* Prevent application from calling me at wrong times */
    if (cinfo->global_state != DSTATE_READY)
        ERREXIT1(cinfo, JERR_BAD_STATE, cinfo->global_state);

#ifdef IDCT_SCALING_SUPPORTED

    /* Compute actual output image dimensions and DCT scaling choices. */
    if (cinfo->scale_num * 8 <= cinfo->scale_denom) {
        /* Provide 1/8 scaling */
        cinfo->output_width = (JDIMENSION)
            jdiv_round_up((long) cinfo->image_width, 8L);
        cinfo->output_height = (JDIMENSION)
            jdiv_round_up((long) cinfo->image_height, 8L);
        cinfo->min_DCT_scaled_size = 1;
    } else if (cinfo->scale_num * 4 <= cinfo->scale_denom) {
        /* Provide 1/4 scaling */
        cinfo->output_width = (JDIMENSION)
            jdiv_round_up((long) cinfo->image_width, 4L);
        cinfo->output_height = (JDIMENSION)
            jdiv_round_up((long) cinfo->image_height, 4L);
        cinfo->min_DCT_scaled_size = 2;
    } else if (cinfo->scale_num * 2 <= cinfo->scale_denom) {
        /* Provide 1/2 scaling */
        cinfo->output_width = (JDIMENSION)
            jdiv_round_up((long) cinfo->image_width, 2L);
        cinfo->output_height = (JDIMENSION)
            jdiv_round_up((long) cinfo->image_height, 2L);
        cinfo->min_DCT_scaled_size = 4;
    } else {
        /* Provide 1/1 scaling */
        cinfo->output_width = cinfo->image_width;
        cinfo->output_height = cinfo->image_height;
        cinfo->min_DCT_scaled_size = DCTSIZE;
    }

    /* In selecting the actual DCT scaling for each component, we try to
     * scale up the chroma components via IDCT scaling rather than upsampling.
     * This saves time if the upsampler gets to use 1:1 scaling.
     * Note this code assumes that the supported DCT scalings are powers of 2.
     */
    for (ci = 0, comp_ptr = cinfo->comp_info; ci < cinfo->num_components;
         ci++, comp_ptr++) {
        int ssize = cinfo->min_DCT_scaled_size;
        while (ssize < DCTSIZE &&
            (comp_ptr->h_samp_factor * ssize * 2 <=
             cinfo->max_h_samp_factor * cinfo->min_DCT_scaled_size) &&
            (comp_ptr->v_samp_factor * ssize * 2 <=
             cinfo->max_v_samp_factor * cinfo->min_DCT_scaled_size)) {
            ssize = ssize * 2;
        }
        comp_ptr->DCT_scaled_size = ssize;
    }

    /* Recompute downsampled dimensions of components;
     * application needs to know these if using raw downsampled data.
     */
    for (ci = 0, comp_ptr = cinfo->comp_info; ci < cinfo->num_components;
         ci++, comp_ptr++) {
        /* Size in samples, after IDCT scaling */
        comp_ptr->downsampled_width = (JDIMENSION)
            jdiv_round_up((long) cinfo->image_width *
                (long) (comp_ptr->h_samp_factor * comp_ptr->DCT_scaled_size),
                (long) (cinfo->max_h_samp_factor * DCTSIZE));
        comp_ptr->downsampled_height = (JDIMENSION)
            jdiv_round_up((long) cinfo->image_height *
                (long) (comp_ptr->v_samp_factor * comp_ptr->DCT_scaled_size),
                (long) (cinfo->max_v_samp_factor * DCTSIZE));
    }

#else /* !IDCT_SCALING_SUPPORTED */

```

```

/* Hardwire it to "no scaling" */
cinfo->output_width = cinfo->image_width;
cinfo->output_height = cinfo->image_height;
/* jinput.c has already initialized DCT_scaled_size to DCTSIZE,
 * and has computed unscaled downsampled_width and downsampled_height.
 */

#endif /* IDCT_SCALING_SUPPORTED */

/* Report number of components in selected colorspace. */
/* Probably this should be in the color conversion module... */
switch (cinfo->out_color_space) {
case JCS_GRAYSCALE:
    cinfo->out_color_components = 1;
    break;
case JCS_RGB:
    #if RGB_PIXELSIZE != 3
        cinfo->out_color_components = RGB_PIXELSIZE;
        break;
    #endif /* else share code with YCbCr */
case JCS_YCbCr:
    cinfo->out_color_components = 3;
    break;
case JCS_CMYK:
case JCS_YCCK:
    cinfo->out_color_components = 4;
    break;
default:
    /* else must be same colorspace as in file */
    cinfo->out_color_components = cinfo->num_components;
    break;
}
cinfo->output_components = (cinfo->quantize_colors ? 1 :
    cinfo->out_color_components);

/* See if upsampler will want to emit more than one row at a time */
if (use_merged_upsample(cinfo))
    cinfo->rec_outbuf_height = cinfo->max_v_samp_factor;
else
    cinfo->rec_outbuf_height = 1;

/*
 * Several decompression processes need to range-limit values to the range
 * 0..MAXJSAMPLE; the input value may fall somewhat outside this range
 * due to noise introduced by quantization, roundoff error, etc. These
 * processes are inner loops and need to be as fast as possible. On most
 * machines, particularly CPUs with pipelines or instruction prefetch,
 * a (subscript-check-less) C table lookup
 *     x = sample_range_limit[x];
 * is faster than explicit tests
 *     if (x < 0) x = 0;
 *     else if (x > MAXJSAMPLE) x = MAXJSAMPLE;
 * These processes all use a common table prepared by the routine below.
 *
 * For most steps we can mathematically guarantee that the initial value
 * of x is within MAXJSAMPLE+1 of the legal range, so a table running from
 * -(MAXJSAMPLE+1) to 2*MAXJSAMPLE+1 is sufficient. But for the initial
 * limiting step (just after the IDCT), a wildly out-of-range value is
 * possible if the input data is corrupt. To avoid any chance of indexing
 * off the end of memory and getting a bad-pointer trap, we perform the
 * post-IDCT limiting thus:
 *     x = range_limit[x & MASK];
 * where MASK is 2 bits wider than legal sample data, ie 10 bits for 8-bit
 * samples. Under normal circumstances this is more than enough range and
 * a correct output will be generated; with bogus input data the mask will
 * cause wraparound, and we will safely generate a bogus-but-in-range output.
 * For the post-IDCT step, we want to convert the data from signed to unsigned
 * representation by adding CENTERJSAMPLE at the same time that we limit it.
 * So the post-IDCT limiting table ends up looking like this:
 *     CENTERJSAMPLE, CENTERJSAMPLE+1, ..., MAXJSAMPLE,
 *     MAXJSAMPLE (repeat 2*(MAXJSAMPLE+1)-CENTERJSAMPLE times),
 *     0 (repeat 2*(MAXJSAMPLE+1)-CENTERJSAMPLE times),
 *     0, 1, ..., CENTERJSAMPLE-1
 * Negative inputs select values from the upper half of the table after
 * masking.
 *
 * We can save some space by overlapping the start of the post-IDCT table
 * with the simpler range limiting table. The post-IDCT table begins at

```

```

* sample_range_limit + CENTERJSAMPLE.
*
* Note that the table is allocated in near data space on PCs; it's small
* enough and used often enough to justify this.
*/

```

```

LOCAL(void)
prepare_range_limit_table (j_decompress_ptr cinfo)
/* Allocate and fill in the sample_range_limit table */
{
    JSAMPLE * table;
    int i;

    table = (JSAMPLE *)
        (*cinfo->mem->alloc_small) ((j_common_ptr) cinfo, JPOOL_IMAGE,
        (5 * (MAXJSAMPLE+1) + CENTERJSAMPLE) * SIZEOF(JSAMPLE));
    table += (MAXJSAMPLE+1); /* allow negative subscripts of simple table */
    cinfo->sample_range_limit = table;
    /* First segment of "simple" table: limit[x] = 0 for x < 0 */
    MEMZERO(table - (MAXJSAMPLE+1), (MAXJSAMPLE+1) * SIZEOF(JSAMPLE));
    /* Main part of "simple" table: limit[x] = x */
    for (i = 0; i <= MAXJSAMPLE; i++)
        table[i] = (JSAMPLE) i;
    table += CENTERJSAMPLE; /* Point to where post-IDCT table starts */
    /* End of simple table, rest of first half of post-IDCT table */
    for (i = CENTERJSAMPLE; i < 2*(MAXJSAMPLE+1); i++)
        table[i] = MAXJSAMPLE;
    /* Second half of post-IDCT table */
    MEMZERO(table + (2 * (MAXJSAMPLE+1)),
        (2 * (MAXJSAMPLE+1) - CENTERJSAMPLE) * SIZEOF(JSAMPLE));
    MEMCOPY(table + (4 * (MAXJSAMPLE+1) - CENTERJSAMPLE),
        cinfo->sample_range_limit, CENTERJSAMPLE * SIZEOF(JSAMPLE));
}

```

```

Master selection of decompression modules.
This is done once at jpeg_start_decompress time. We determine
which modules will be used and give them appropriate initialization calls.
We also initialize the decompressor input side to begin consuming data.

```

```

Since jpeg_read_header has finished, we know what is in the SOF
and (first) SOS markers. We also have all the application parameter
settings.
*/

```

```

LOCAL(void)
master_selection (j_decompress_ptr cinfo)
{
    my_master_ptr master = (my_master_ptr) cinfo->master;
    boolean use_c_buffer;
    long samplesperrow;
    JDIMENSION jd_samplesperrow;

    /* Initialize dimensions and other stuff */
    jpeg_calc_output_dimensions(cinfo);
    prepare_range_limit_table(cinfo);

    /* Width of an output scanline must be representable as JDIMENSION. */
    samplesperrow = (long) cinfo->output_width * (long) cinfo->out_color_components;
    jd_samplesperrow = (JDIMENSION) samplesperrow;
    if ((long) jd_samplesperrow != samplesperrow)
        ERREXIT(cinfo, JERR_WIDTH_OVERFLOW);

    /* Initialize my private state */
    master->pass_number = 0;
    master->using_merged_upsample = use_merged_upsample(cinfo);

    /* Color quantizer selection */
    master->quantizer_1pass = NULL;
    master->quantizer_2pass = NULL;
    /* No mode changes if not using buffered-image mode. */
    if (! cinfo->quantize_colors || ! cinfo->buffered_image) {
        cinfo->enable_1pass_quant = FALSE;
        cinfo->enable_external_quant = FALSE;
        cinfo->enable_2pass_quant = FALSE;
    }
    if (cinfo->quantize_colors) {
        if (cinfo->raw_data_out)
            ERREXIT(cinfo, JERR_NOTIMPL);
    }
}

```

```

/* 2-pass quantizer only works in 3-component color space. */
if (cinfo->out_color_components != 3) {
    cinfo->enable_1pass_quant = TRUE;
    cinfo->enable_external_quant = FALSE;
    cinfo->enable_2pass_quant = FALSE;
    cinfo->colormap = NULL;
} else if (cinfo->colormap != NULL) {
    cinfo->enable_external_quant = TRUE;
} else if (cinfo->two_pass_quantize) {
    cinfo->enable_2pass_quant = TRUE;
} else {
    cinfo->enable_1pass_quant = TRUE;
}

if (cinfo->enable_1pass_quant) {
#ifdef QUANT_1PASS_SUPPORTED
    jinit_1pass_quantizer(cinfo);
    master->quantizer_1pass = cinfo->cquantize;
#else
    ERREXIT(cinfo, JERR_NOT_COMPILED);
#endif
}

/* We use the 2-pass code to map to external colormaps. */
if (cinfo->enable_2pass_quant || cinfo->enable_external_quant) {
#ifdef QUANT_2PASS_SUPPORTED
    jinit_2pass_quantizer(cinfo);
    master->quantizer_2pass = cinfo->cquantize;
#else
    ERREXIT(cinfo, JERR_NOT_COMPILED);
#endif
}

/* If both quantizers are initialized, the 2-pass one is left active;
 * this is necessary for starting with quantization to an external map.
 */

/* Post-processing: in particular, color conversion first */
if (!cinfo->raw_data_out) {
    if (master->using_merged_upsample) {
#ifdef UPSAMPLE_MERGING_SUPPORTED
        jinit_merged_upsampler(cinfo); /* does color conversion too */
    #else
        ERREXIT(cinfo, JERR_NOT_COMPILED);
    #endif
    } else {
        jinit_color_deconverter(cinfo);
        jinit_upsampler(cinfo);
    }
    jinit_d_post_controller(cinfo, cinfo->enable_2pass_quant);
}

/* Inverse DCT */
jinit_inverse_dct(cinfo);
/* Entropy decoding: either Huffman or arithmetic coding. */
if (cinfo->arith_code) {
    ERREXIT(cinfo, JERR_ARITH_NOTIMPL);
} else {
    if (cinfo->progressive_mode) {
#ifdef D_PROGRESSIVE_SUPPORTED
        jinit_phuff_decoder(cinfo);
    #else
        ERREXIT(cinfo, JERR_NOT_COMPILED);
    #endif
    } else {
        jinit_huff_decoder(cinfo);
    }
}

/* Initialize principal buffer controllers. */
use_c_buffer = cinfo->inputctl->has_multiple_scans || cinfo->buffered_image;
jinit_d_coef_controller(cinfo, use_c_buffer);

if (!cinfo->raw_data_out)
    jinit_d_main_controller(cinfo, FALSE /* never need full buffer here */);

/* We can now tell the memory manager to allocate virtual arrays. */
(*cinfo->mem->realize_virt_arrays) ((j_common_ptr) cinfo);

/* Initialize input side of decompressor to consume first scan. */
(*cinfo->inputctl->start_input_pass) (cinfo);

```

```

#ifdef D_MULTISCAN_FILES_SUPPORTED
/* If jpeg_start_decompress will read the whole file, initialize
 * progress monitoring appropriately. The input step is counted
 * as one pass.
 */
if (cinfo->progress != NULL && ! cinfo->buffered_image &&
    cinfo->inputctl->has_multiple_scans) {
    int nscans;
    /* Estimate number of scans to set pass_limit. */
    if (cinfo->progressive_mode) {
        /* Arbitrarily estimate 2 interleaved DC scans + 3 AC scans/component. */
        nscans = 2 + 3 * cinfo->num_components;
    } else {
        /* For a nonprogressive multiscan file, estimate 1 scan per component. */
        nscans = cinfo->num_components;
    }
    cinfo->progress->pass_counter = 0L;
    cinfo->progress->pass_limit = (long) cinfo->total_iMCU_rows * nscans;
    cinfo->progress->completed_passes = 0;
    cinfo->progress->total_passes = (cinfo->enable_2pass_quant ? 3 : 2);
    /* Count the input pass as done */
    master->pass_number++;
}
#endif /* D_MULTISCAN_FILES_SUPPORTED */

/*
 * Per-pass setup.
 * This is called at the beginning of each output pass. We determine which
 * modules will be active during this pass and give them appropriate
 * start_pass calls. We also set is_dummy_pass to indicate whether this
 * is a "real" output pass or a dummy pass for color quantization.
 * (In the latter case, jdapistd.c will crank the pass to completion.)
 */

METHODDEF(void)
prepare_for_output_pass (j_decompress_ptr cinfo)
{
    my_master_ptr master = (my_master_ptr) cinfo->master;

    if (master->pub.is_dummy_pass) {
#ifdef QUANT_2PASS_SUPPORTED
        /* Final pass of 2-pass quantization */
        master->pub.is_dummy_pass = FALSE;
        (*cinfo->cquantize->start_pass) (cinfo, FALSE);
        (*cinfo->post->start_pass) (cinfo, JBUF_CRANK_DEST);
        (*cinfo->main->start_pass) (cinfo, JBUF_CRANK_DEST);
    #else
        ERREXIT(cinfo, JERR_NOT_COMPILED);
    #endif /* QUANT_2PASS_SUPPORTED */
    } else {
        if (cinfo->quantize_colors && cinfo->colormap == NULL) {
            /* Select new quantization method */
            if (cinfo->two_pass_quantize && cinfo->enable_2pass_quant) {
                cinfo->cquantize = master->quantizer_2pass;
                master->pub.is_dummy_pass = TRUE;
            } else if (cinfo->enable_1pass_quant) {
                cinfo->cquantize = master->quantizer_1pass;
            } else {
                ERREXIT(cinfo, JERR_MODE_CHANGE);
            }
        }
        (*cinfo->idct->start_pass) (cinfo);
        (*cinfo->coef->start_output_pass) (cinfo);
        if (! cinfo->raw_data_out) {
            if (! master->using_merged_upsample)
                (*cinfo->cconvert->start_pass) (cinfo);
            (*cinfo->upsample->start_pass) (cinfo);
            if (cinfo->quantize_colors)
                (*cinfo->cquantize->start_pass) (cinfo, master->pub.is_dummy_pass);
            (*cinfo->post->start_pass) (cinfo,
                (master->pub.is_dummy_pass ? JBUF_SAVE_AND_PASS : JBUF_PASS_THRU));
            (*cinfo->main->start_pass) (cinfo, JBUF_PASS_THRU);
        }
    }

    /* Set up progress monitor's pass info if present */
    if (cinfo->progress != NULL) {
        cinfo->progress->completed_passes = master->pass_number;
    }
}

```



```

    cinfo->progress->total_passes = master->pass_number +
        (master->pub.is_dummy_pass ? 2 : 1);
    /* In buffered-image mode, we assume one more output pass if EOI has not
     * yet reached, but no more passes if EOI has been reached.
     */
    if (cinfo->buffered_image && ! cinfo->inputctl->eoi_reached) {
        cinfo->progress->total_passes += (cinfo->enable_2pass_quant ? 2 : 1);
    }
}

```

```

/*
 * Finish up at end of an output pass.
 */

```

```

METHODDEF(void)
finish_output_pass (j_decompress_ptr cinfo)
{
    my_master_ptr master = (my_master_ptr) cinfo->master;

    if (cinfo->quantize_colors)
        (*cinfo->cquantize->finish_pass) (cinfo);
    master->pass_number++;
}

```

```

#ifdef D_MULTISCAN_FILES_SUPPORTED

```

```

/*
 * Switch to a new external colormap between output passes.
 */

```

```

GLOBAL(void)
jpeg_new_colormap (j_decompress_ptr cinfo)
{
    my_master_ptr master = (my_master_ptr) cinfo->master;

    /* Prevent application from calling me at wrong times */
    if (cinfo->global_state != DSTATE_BUFIMAGE)
        ERREXIT1(cinfo, JERR_BAD_STATE, cinfo->global_state);

    if (cinfo->quantize_colors && cinfo->enable_external_quant &&
        cinfo->colormap != NULL) {
        /* Select 2-pass quantizer for external colormap use */
        cinfo->cquantize = master->quantizer_2pass;
        /* Notify quantizer of colormap change */
        (*cinfo->cquantize->new_color_map) (cinfo);
        master->pub.is_dummy_pass = FALSE; /* just in case */
    } else
        ERREXIT(cinfo, JERR_MODE_CHANGE);
}

```

```

#endif /* D_MULTISCAN_FILES_SUPPORTED */

```

```

/*
 * Initialize master decompression control and select active modules.
 * This is performed at the start of jpeg_start_decompress.
 */

```

```

GLOBAL(void)
jinit_master_decompress (j_decompress_ptr cinfo)
{
    my_master_ptr master;

    master = (my_master_ptr)
        (*cinfo->mem->alloc_small) ((j_common_ptr) cinfo, JPOOL_IMAGE,
            sizeof(my_decomp_master));
    cinfo->master = (struct jpeg_decomp_master *) master;
    master->pub.prepare_for_output_pass = prepare_for_output_pass;
    master->pub.finish_output_pass = finish_output_pass;

    master->pub.is_dummy_pass = FALSE;

    master_selection(cinfo);
}

```

```

/*
 * jdmerge.c
 *
 * Copyright (C) 1994-1996, Thomas G. Lane.
 * This file is part of the Independent JPEG Group's software.
 * For conditions of distribution and use, see the accompanying README file.
 *
 * This file contains code for merged upsampling/color conversion.
 *
 * This file combines functions from jdsample.c and jdcolor.c;
 * read those files first to understand what's going on.
 *
 * When the chroma components are to be upsampled by simple replication
 * (ie, box filtering), we can save some work in color conversion by
 * calculating all the output pixels corresponding to a pair of chroma
 * samples at one time. In the conversion equations
 *   R = Y          + K1 * Cr
 *   G = Y + K2 * Cb + K3 * Cr
 *   B = Y + K4 * Cb
 * only the Y term varies among the group of pixels corresponding to a pair
 * of chroma samples, so the rest of the terms can be calculated just once.
 * At typical sampling ratios, this eliminates half or three-quarters of the
 * multiplications needed for color conversion.
 *
 * This file currently provides implementations for the following cases:
 *   YCbCr => RGB color conversion only.
 *   Sampling ratios of 2h1v or 2h2v.
 *   No scaling needed at upsample time.
 *   Corner-aligned (non-CCIR601) sampling alignment.
 * Other special cases could be added, but in most applications these are
 * the only common cases. (For uncommon cases we fall back on the more
 * general code in jdsample.c and jdcolor.c.)
 */

#define JPEG_INTERNALS
#include "jinclude.h"
#include "jpeglib.h"

#ifdef UPSAMPLE_MERGING_SUPPORTED

/* Private subobject */

typedef struct {
  struct jpeg_upsampler pub; /* public fields */

  /* Pointer to routine to do actual upsampling/conversion of one row group */
  METHODDEF(void, upmethod, (j_decompress_ptr cinfo,
    JSAMPIMAGE input_buf, JDIMENSION in_row_group_ctr,
    JSAMPARRAY output_buf));

  /* Private state for YCC->RGB conversion */
  int * Cr_r_tab; /* => table for Cr to R conversion */
  int * Cb_b_tab; /* => table for Cb to B conversion */
  INT32 * Cr_g_tab; /* => table for Cr to G conversion */
  INT32 * Cb_g_tab; /* => table for Cb to G conversion */

  /* For 2:1 vertical sampling, we produce two output rows at a time.
   * We need a "spare" row buffer to hold the second output row if the
   * application provides just a one-row buffer; we also use the spare
   * to discard the dummy last row if the image height is odd.
   */
  JSAMPROW spare_row;
  boolean spare_full; /* T if spare buffer is occupied */

  JDIMENSION out_row_width; /* samples per output row */
  JDIMENSION rows_to_go; /* counts rows remaining in image */
} my_upsampler;

typedef my_upsampler * my_upsample_ptr;

#define SCALEBITS 16 /* speediest right-shift on some machines */
#define ONE_HALF ((INT32) 1 << (SCALEBITS-1))
#define FIX(x) ((INT32) ((x) * (1L<<SCALEBITS) + 0.5))

/*
 * Initialize tables for YCC->RGB colorspace conversion.
 * This is taken directly from jdcolor.c; see that file for more info.
 */

```

```

LOCAL(void)
build_ycc_rgb_table (j_decompress_ptr cinfo)
{
    my_upsample_ptr upsample = (my_upsample_ptr) cinfo->upsample;
    int i;
    INT32 x;
    SHIFT_TEMPS

    upsample->Cr_r_tab = (int *)
        (*cinfo->mem->alloc_small) ((j_common_ptr) cinfo, JPOOL_IMAGE,
            (MAXJSAMPLE+1) * SIZEOF(int));
    upsample->Cb_b_tab = (int *)
        (*cinfo->mem->alloc_small) ((j_common_ptr) cinfo, JPOOL_IMAGE,
            (MAXJSAMPLE+1) * SIZEOF(int));
    upsample->Cr_g_tab = (INT32 *)
        (*cinfo->mem->alloc_small) ((j_common_ptr) cinfo, JPOOL_IMAGE,
            (MAXJSAMPLE+1) * SIZEOF(INT32));
    upsample->Cb_g_tab = (INT32 *)
        (*cinfo->mem->alloc_small) ((j_common_ptr) cinfo, JPOOL_IMAGE,
            (MAXJSAMPLE+1) * SIZEOF(INT32));

    for (i = 0, x = -CENTERJSAMPLE; i <= MAXJSAMPLE; i++, x++) {
        /* i is the actual input pixel value, in the range 0..MAXJSAMPLE */
        /* The Cb or Cr value we are thinking of is x = i - CENTERJSAMPLE */
        /* Cr=>R value is nearest int to 1.40200 * x */
        upsample->Cr_r_tab[i] = (int)
            RIGHT_SHIFT(FIX(1.40200) * x + ONE_HALF, SCALEBITS);
        /* Cb=>B value is nearest int to 1.77200 * x */
        upsample->Cb_b_tab[i] = (int)
            RIGHT_SHIFT(FIX(1.77200) * x + ONE_HALF, SCALEBITS);
        /* Cr=>G value is scaled-up -0.71414 * x */
        upsample->Cr_g_tab[i] = (- FIX(0.71414)) * x;
        /* Cb=>G value is scaled-up -0.34414 * x */
        /* We also add in ONE_HALF so that need not do it in inner loop */
        upsample->Cb_g_tab[i] = (- FIX(0.34414)) * x + ONE_HALF;
    }
}

```

```

/* Initialize for an upsampling pass.
 */

```

```

METHODDEF(void)
start_pass_merged_upsample (j_decompress_ptr cinfo)
{
    my_upsample_ptr upsample = (my_upsample_ptr) cinfo->upsample;

    /* Mark the spare buffer empty */
    upsample->spare_full = FALSE;
    /* Initialize total-height counter for detecting bottom of image */
    upsample->rows_to_go = cinfo->output_height;
}

```

```

/*
 * Control routine to do upsampling (and color conversion).
 *
 * The control routine just handles the row buffering considerations.
 */

```

```

METHODDEF(void)
merged_2v_upsample (j_decompress_ptr cinfo,
    JSAMPIMAGE input_buf, JDIMENSION *in_row_group_ctr,
    JDIMENSION in_row_groups_avail,
    JSAMPARRAY output_buf, JDIMENSION *out_row_ctr,
    JDIMENSION out_rows_avail)
/* 2:1 vertical sampling case: may need a spare row. */
{
    my_upsample_ptr upsample = (my_upsample_ptr) cinfo->upsample;
    JSAMPROW work_ptrs[2];
    JDIMENSION num_rows; /* number of rows returned to caller */

    if (upsample->spare_full) {
        /* If we have a spare row saved from a previous cycle, just return it. */
        jcopy_sample_rows(& upsample->spare_row, 0, output_buf + *out_row_ctr, 0,
            1, upsample->out_row_width);
        num_rows = 1;
        upsample->spare_full = FALSE;
    }
}

```

```

) else {
    /* Figure number of rows to return to caller. */
    num_rows = 2;
    /* Not more than the distance to the end of the image. */
    if (num_rows > upsample->rows_to_go)
        num_rows = upsample->rows_to_go;
    /* And not more than what the client can accept: */
    out_rows_avail -= *out_row_ctr;
    if (num_rows > out_rows_avail)
        num_rows = out_rows_avail;
    /* Create output pointer array for upsampler. */
    work_ptrs[0] = output_buf[*out_row_ctr];
    if (num_rows > 1) {
        work_ptrs[1] = output_buf[*out_row_ctr + 1];
    } else {
        work_ptrs[1] = upsample->spare_row;
        upsample->spare_full = TRUE;
    }
    /* Now do the upsampling. */
    (*upsample->upmethod) (cinfo, input_buf, *in_row_group_ctr, work_ptrs);
}

/* Adjust counts */
*out_row_ctr += num_rows;
upsample->rows_to_go -= num_rows;
/* When the buffer is emptied, declare this input row group consumed */
if (! upsample->spare_full)
    (*in_row_group_ctr)++;
}

METHODDEF(void)
merged_lv_upsample (j_decompress_ptr cinfo,
                    JSAMPIMAGE input_buf, JDIMENSION *in_row_group_ctr,
                    JDIMENSION in_row_groups_avail,
                    JSAMPARRAY output_buf, JDIMENSION *out_row_ctr,
                    JDIMENSION out_rows_avail)
/* 1:1 vertical sampling case: much easier, never need a spare row. */
{
    my_upsample_ptr upsample = (my_upsample_ptr) cinfo->upsample;

    /* Just do the upsampling. */
    (*upsample->upmethod) (cinfo, input_buf, *in_row_group_ctr,
                          output_buf + *out_row_ctr);
    /* Adjust counts */
    (*out_row_ctr)++;
    (*in_row_group_ctr)++;
}

/*
 * These are the routines invoked by the control routines to do
 * the actual upsampling/conversion. One row group is processed per call.
 *
 * Note: since we may be writing directly into application-supplied buffers,
 * we have to be honest about the output width; we can't assume the buffer
 * has been rounded up to an even width.
 */

/*
 * Upsample and color convert for the case of 2:1 horizontal and 1:1 vertical.
 */

METHODDEF(void)
h2v1_merged_upsample (j_decompress_ptr cinfo,
                      JSAMPIMAGE input_buf, JDIMENSION in_row_group_ctr,
                      JSAMPARRAY output_buf)
{
    my_upsample_ptr upsample = (my_upsample_ptr) cinfo->upsample;
    register int y, cred, cgreen, cblue;
    int cb, cr;
    register JSAMPROW outptr;
    JSAMPROW inptr0, inptr1, inptr2;
    JDIMENSION col;
    /* copy these pointers into registers if possible */
    register JSAMPLE * range_limit = cinfo->sample_range_limit;
    int * Crrtab = upsample->Cr_r_tab;
    int * Cbftab = upsample->Cb_b_tab;
    INT32 * Crgtab = upsample->Cr_g_tab;

```

```

INT32 * Cbgtab = upsample->Cr_g_tab;
SHIFT_TEMPS

inptr0 = input_buf[0][in_row_group_ctr];
inptr1 = input_buf[1][in_row_group_ctr];
inptr2 = input_buf[2][in_row_group_ctr];
outptr = output_buf[0];
/* Loop for each pair of output pixels */
for (col = cinfo->output_width >> 1; col > 0; col--) {
    /* Do the chroma part of the calculation */
    cb = GETJSAMPLE(*inptr1++);
    cr = GETJSAMPLE(*inptr2++);
    cred = Crrtab[cr];
    cgreen = (int) RIGHT_SHIFT(Cbgtab[cb] + Crgtab[cr], SCALEBITS);
    cblue = Cbbtab[cb];
    /* Fetch 2 Y values and emit 2 pixels */
    y = GETJSAMPLE(*inptr0++);
    outptr[RGB_RED] = range_limit[y + cred];
    outptr[RGB_GREEN] = range_limit[y + cgreen];
    outptr[RGB_BLUE] = range_limit[y + cblue];
    outptr += RGB_PIXELSIZE;
    y = GETJSAMPLE(*inptr0++);
    outptr[RGB_RED] = range_limit[y + cred];
    outptr[RGB_GREEN] = range_limit[y + cgreen];
    outptr[RGB_BLUE] = range_limit[y + cblue];
    outptr += RGB_PIXELSIZE;
}
/* If image width is odd, do the last output column separately */
if (cinfo->output_width & 1) {
    cb = GETJSAMPLE(*inptr1);
    cr = GETJSAMPLE(*inptr2);
    cred = Crrtab[cr];
    cgreen = (int) RIGHT_SHIFT(Cbgtab[cb] + Crgtab[cr], SCALEBITS);
    cblue = Cbbtab[cb];
    y = GETJSAMPLE(*inptr0);
    outptr[RGB_RED] = range_limit[y + cred];
    outptr[RGB_GREEN] = range_limit[y + cgreen];
    outptr[RGB_BLUE] = range_limit[y + cblue];
}
/*
 * Upsample and color convert for the case of 2:1 horizontal and 2:1 vertical.
 */
METHODDEF(void)
h2v2_merged_upsample (j_decompress_ptr cinfo,
                      JSAMPIMAGE input_buf, JDIMENSION in_row_group_ctr,
                      JSAMPARRAY output_buf)
{
    my_upsample_ptr upsample = (my_upsample_ptr) cinfo->upsample;
    register int y, cred, cgreen, cblue;
    int cb, cr;
    register JSAMPROW outptr0, outptr1;
    JSAMPROW inptr00, inptr01, inptr1, inptr2;
    JDIMENSION col;
    /* copy these pointers into registers if possible */
    register JSAMPLE * range_limit = cinfo->sample_range_limit;
    int * Crrtab = upsample->Cr_r_tab;
    int * Cbbtab = upsample->Cb_b_tab;
    INT32 * Crgtab = upsample->Cr_g_tab;
    INT32 * Cbgtab = upsample->Cb_g_tab;
    SHIFT_TEMPS

    inptr00 = input_buf[0][in_row_group_ctr*2];
    inptr01 = input_buf[0][in_row_group_ctr*2 + 1];
    inptr1 = input_buf[1][in_row_group_ctr];
    inptr2 = input_buf[2][in_row_group_ctr];
    outptr0 = output_buf[0];
    outptr1 = output_buf[1];
    /* Loop for each group of output pixels */
    for (col = cinfo->output_width >> 1; col > 0; col--) {
        /* Do the chroma part of the calculation */
        cb = GETJSAMPLE(*inptr1++);
        cr = GETJSAMPLE(*inptr2++);
        cred = Crrtab[cr];
        cgreen = (int) RIGHT_SHIFT(Cbgtab[cb] + Crgtab[cr], SCALEBITS);
        cblue = Cbbtab[cb];
        /* Fetch 4 Y values and emit 4 pixels */

```

```

y = GETJSAMPLE(*inptr0++);
outptr0[RGB_RED] = range_limit[y + cred];
outptr0[RGB_GREEN] = range_limit[y + cgreen];
outptr0[RGB_BLUE] = range_limit[y + cblue];
outptr0 += RGB_PIXELSIZE;
y = GETJSAMPLE(*inptr0++);
outptr0[RGB_RED] = range_limit[y + cred];
outptr0[RGB_GREEN] = range_limit[y + cgreen];
outptr0[RGB_BLUE] = range_limit[y + cblue];
outptr0 += RGB_PIXELSIZE;
y = GETJSAMPLE(*inptr01++);
outptr1[RGB_RED] = range_limit[y + cred];
outptr1[RGB_GREEN] = range_limit[y + cgreen];
outptr1[RGB_BLUE] = range_limit[y + cblue];
outptr1 += RGB_PIXELSIZE;
y = GETJSAMPLE(*inptr01++);
outptr1[RGB_RED] = range_limit[y + cred];
outptr1[RGB_GREEN] = range_limit[y + cgreen];
outptr1[RGB_BLUE] = range_limit[y + cblue];
outptr1 += RGB_PIXELSIZE;
}
/* If image width is odd, do the last output column separately */
if (cinfo->output_width & 1) {
    cb = GETJSAMPLE(*inptr1);
    cr = GETJSAMPLE(*inptr2);
    cred = Crrtab[cr];
    cgreen = (int) RIGHT_SHIFT(Cbgtab[cb] + Crgtab[cr], SCALEBITS);
    cblue = Cbbtab[cb];
    y = GETJSAMPLE(*inptr00);
    outptr0[RGB_RED] = range_limit[y + cred];
    outptr0[RGB_GREEN] = range_limit[y + cgreen];
    outptr0[RGB_BLUE] = range_limit[y + cblue];
    y = GETJSAMPLE(*inptr01);
    outptr1[RGB_RED] = range_limit[y + cred];
    outptr1[RGB_GREEN] = range_limit[y + cgreen];
    outptr1[RGB_BLUE] = range_limit[y + cblue];
}
}

/*
 * Module initialization routine for merged upsampling/color conversion.
 */
/* NB: this is called under the conditions determined by use_merged_upsample()
 * in jdmaster.c. That routine MUST correspond to the actual capabilities
 * of this module; no safety checks are made here.
 */
GLOBAL(void)
jinit_merged_upsampler (j_decompress_ptr cinfo)
{
    my_upsample_ptr upsample;

    upsample = (my_upsample_ptr)
        (*cinfo->mem->alloc_small) ((j_common_ptr) cinfo, JPOOL_IMAGE,
            SIZEOF(my_upsampler));
    cinfo->upsample = (struct jpeg_upsampler *) upsample;
    upsample->pub.start_pass = start_pass_merged_upsample;
    upsample->pub.need_context_rows = FALSE;

    upsample->out_row_width = cinfo->output_width * cinfo->out_color_components;

    if (cinfo->max_v_samp_factor == 2) {
        upsample->pub.upsample = merged_2v_upsample;
        upsample->upmethod = h2v2_merged_upsample;
        /* Allocate a spare row buffer */
        upsample->spare_row = (JSAMPROW)
            (*cinfo->mem->alloc_large) ((j_common_ptr) cinfo, JPOOL_IMAGE,
                (size_t) (upsample->out_row_width * SIZEOF(JSAMPLE)));
    } else {
        upsample->pub.upsample = merged_1v_upsample;
        upsample->upmethod = h2v1_merged_upsample;
        /* No spare row needed */
        upsample->spare_row = NULL;
    }

    build_ycc_rgb_table(cinfo);
}

#endif /* UPSAMPLE_MERGING_SUPPORTED */

```

```

/*
 * jdphuff.c
 *
 * Copyright (C) 1995-1997, Thomas G. Lane.
 * This file is part of the Independent JPEG Group's software.
 * For conditions of distribution and use, see the accompanying README file.
 *
 * This file contains Huffman entropy decoding routines for progressive JPEG.
 *
 * Much of the complexity here has to do with supporting input suspension.
 * If the data source module demands suspension, we want to be able to back
 * up to the start of the current MCU. To do this, we copy state variables
 * into local working storage, and update them back to the permanent
 * storage only upon successful completion of an MCU.
 */

#define JPEG_INTERNALS
#include "jinclude.h"
#include "jpeglib.h"
#include "jdphuff.h" /* Declarations shared with jdphuff.c */

#ifdef D_PROGRESSIVE_SUPPORTED

/*
 * Expanded entropy decoder object for progressive Huffman decoding.
 *
 * The savable_state subrecord contains fields that change within an MCU,
 * but must not be updated permanently until we complete the MCU.
 */

typedef struct {
  unsigned int EOBRUN; /* remaining EOBs in EOBRUN */
  int last_dc_val[MAX_COMPS_IN_SCAN]; /* last DC coef for each component */
} savable_state;

/* This macro is to work around compilers with missing or broken
 * structure assignment. You'll need to fix this code if you have
 * such a compiler and you change MAX_COMPS_IN_SCAN.
 */
#ifdef NO_STRUCT_ASSIGN
#define ASSIGN_STATE(dest,src) ((dest) = (src))
#else
#define ASSIGN_STATE(dest,src) \
  if (MAX_COMPS_IN_SCAN == 4) \
    ((dest).EOBRUN = (src).EOBRUN, \
     (dest).last_dc_val[0] = (src).last_dc_val[0], \
     (dest).last_dc_val[1] = (src).last_dc_val[1], \
     (dest).last_dc_val[2] = (src).last_dc_val[2], \
     (dest).last_dc_val[3] = (src).last_dc_val[3])
#endif

#define NO_STRUCT_ASSIGN
#define ASSIGN_STATE(dest,src) ((dest) = (src))
#else
#define ASSIGN_STATE(dest,src) \
  if (MAX_COMPS_IN_SCAN == 4) \
    ((dest).EOBRUN = (src).EOBRUN, \
     (dest).last_dc_val[0] = (src).last_dc_val[0], \
     (dest).last_dc_val[1] = (src).last_dc_val[1], \
     (dest).last_dc_val[2] = (src).last_dc_val[2], \
     (dest).last_dc_val[3] = (src).last_dc_val[3])
#endif

typedef struct {
  struct jpeg_entropy_decoder pub; /* public fields */

  /* These fields are loaded into local variables at start of each MCU.
   * In case of suspension, we exit WITHOUT updating them.
   */
  bitread_perm_state bitstate; /* Bit buffer at start of MCU */
  savable_state saved; /* Other state at start of MCU */

  /* These fields are NOT loaded into local working state. */
  unsigned int restarts_to_go; /* MCUs left in this restart interval */

  /* Pointers to derived tables (these workspaces have image lifespan) */
  d_derived_tbl * derived_tbls[NUM_HUFF_TBLS];

  d_derived_tbl * ac_derived_tbl; /* active table during an AC scan */
} phuff_entropy_decoder;

typedef phuff_entropy_decoder * phuff_entropy_ptr;

/* Forward declarations */
METHODDEF(boolean) decode_mcu_DC_first JPP((j_decompress_ptr cinfo,
                                           JBLOCKROW *MCU_data));
METHODDEF(boolean) decode_mcu_AC_first JPP((j_decompress_ptr cinfo,
                                           JBLOCKROW *MCU_data));

```

```

METHODDEF(boolean) decode_mcu_refine JPP((j_decompress_ptr cinfo,
                                           JBLOCKROW *MCU_data));
METHODDEF(boolean) decode_mcu_refine JPP((j_decompress_ptr cinfo,
                                           JBLOCKROW *MCU_data));

```

```

/*
 * Initialize for a Huffman-compressed scan.
 */

```

```

METHODDEF(void)
start_pass_phuff_decoder (j_decompress_ptr cinfo)
{
    phuff_entropy_ptr entropy = (phuff_entropy_ptr) cinfo->entropy;
    boolean is_DC_band, bad;
    int ci, coefi, tbl;
    int *coef_bit_ptr;
    jpeg_component_info * comp_ptr;

    is_DC_band = (cinfo->Ss == 0);

    /* Validate scan parameters */
    bad = FALSE;
    if (is_DC_band) {
        if (cinfo->Se != 0)
            bad = TRUE;
    } else {
        /* need not check Ss/Se < 0 since they came from unsigned bytes */
        if (cinfo->Ss > cinfo->Se || cinfo->Se >= DCTSIZE2)
            bad = TRUE;
        /* AC scans may have only one component */
        if (cinfo->comps_in_scan != 1)
            bad = TRUE;
        if (cinfo->Ah != 0) {
            /* Successive approximation refinement scan: must have A1 = Ah-1. */
            if (cinfo->A1 != cinfo->Ah-1)
                bad = TRUE;
        }
        if (cinfo->A1 > 13) /* need not check for < 0 */
            bad = TRUE;
        /* Arguably the maximum A1 value should be less than 13 for 8-bit precision,
         * but the spec doesn't say so, and we try to be liberal about what we
         * accept. Note: large A1 values could result in out-of-range DC
         * coefficients during early scans, leading to bizarre displays due to
         * overflows in the IDCT math. But we won't crash.
         */
        if (bad)
            ERREXIT4(cinfo, JERR_BAD_PROGRESSION,
                    cinfo->Ss, cinfo->Se, cinfo->Ah, cinfo->A1);
        /* Update progression status, and verify that scan order is legal.
         * Note that inter-scan inconsistencies are treated as warnings
         * not fatal errors ... not clear if this is right way to behave.
         */
        for (ci = 0; ci < cinfo->comps_in_scan; ci++) {
            int cindex = cinfo->cur_comp_info[ci]->component_index;
            coef_bit_ptr = & cinfo->coef_bits[cindex][0];
            if (!is_DC_band && coef_bit_ptr[0] < 0) /* AC without prior DC scan */
                WARNMS2(cinfo, JWRN_BOGUS_PROGRESSION, cindex, 0);
            for (coefi = cinfo->Ss; coefi <= cinfo->Se; coefi++) {
                int expected = (coef_bit_ptr[coefi] < 0) ? 0 : coef_bit_ptr[coefi];
                if (cinfo->Ah != expected)
                    WARNMS2(cinfo, JWRN_BOGUS_PROGRESSION, cindex, coefi);
                coef_bit_ptr[coefi] = cinfo->A1;
            }
        }

        /* Select MCU decoding routine */
        if (cinfo->Ah == 0) {
            if (is_DC_band)
                entropy->pub.decode_mcu = decode_mcu_DC_first;
            else
                entropy->pub.decode_mcu = decode_mcu_AC_first;
        } else {
            if (is_DC_band)
                entropy->pub.decode_mcu = decode_mcu_DC_refine;
            else
                entropy->pub.decode_mcu = decode_mcu_AC_refine;
        }
    }
}

```



```

for (ci = 0; ci < cinfo->com_in_scan; ci++) {
    compptr = cinfo->cur_comp_ptr[ci];
    /* Make sure requested tables are present, and compute derived tables.
     * We may build same derived table more than once, but it's not expensive.
     */
    if (is_DC_band) {
        if (cinfo->Ah == 0) { /* DC refinement needs no table */
            tbl = compptr->dc_tbl_no;
            jpeg_make_d_derived_tbl(cinfo, TRUE, tbl,
                                   & entropy->derived_tbls[tbl]);
        }
    } else {
        tbl = compptr->ac_tbl_no;
        jpeg_make_d_derived_tbl(cinfo, FALSE, tbl,
                                & entropy->derived_tbls[tbl]);
        /* remember the single active table */
        entropy->ac_derived_tbl = entropy->derived_tbls[tbl];
    }
    /* Initialize DC predictions to 0 */
    entropy->saved.last_dc_val[ci] = 0;
}

/* Initialize bitread state variables */
entropy->bitstate.bits_left = 0;
entropy->bitstate.get_buffer = 0; /* unnecessary, but keeps Purify quiet */
entropy->pub.insufficient_data = FALSE;

/* Initialize private state variables */
entropy->saved.EOBRUN = 0;

/* Initialize restart counter */
entropy->restarts_to_go = cinfo->restart_interval;
}

```

Figure F.12: extend sign bit.

\* On some machines, a shift and add will be faster than a table lookup.

```

#ifdef AVOID_TABLES
#define HUFF_EXTEND(x,s) ((x) < (1<<((s)-1)) ? (x) + (((-1)<<(s)) + 1) : (x))
#else
#define HUFF_EXTEND(x,s) ((x) < extend_test[s] ? (x) + extend_offset[s] : (x))
static const int extend_test[16] = /* entry n is 2**(n-1) */
{ 0, 0x0001, 0x0002, 0x0004, 0x0008, 0x0010, 0x0020, 0x0040, 0x0080,
  0x0100, 0x0200, 0x0400, 0x0800, 0x1000, 0x2000, 0x4000 };
static const int extend_offset[16] = /* entry n is (-1 << n) + 1 */
{ 0, ((-1)<<1) + 1, ((-1)<<2) + 1, ((-1)<<3) + 1, ((-1)<<4) + 1,
  ((-1)<<5) + 1, ((-1)<<6) + 1, ((-1)<<7) + 1, ((-1)<<8) + 1,
  ((-1)<<9) + 1, ((-1)<<10) + 1, ((-1)<<11) + 1, ((-1)<<12) + 1,
  ((-1)<<13) + 1, ((-1)<<14) + 1, ((-1)<<15) + 1 };
#endif /* AVOID_TABLES */

```

```

/*
 * Check for a restart marker & resynchronize decoder.
 * Returns FALSE if must suspend.
 */
LOCAL(boolean)
process_restart (j_decompress_ptr cinfo)
{
    phuff_entropy_ptr entropy = (phuff_entropy_ptr) cinfo->entropy;
    int ci;

    /* Throw away any unused bits remaining in bit buffer; */
    /* include any full bytes in next_marker's count of discarded bytes */
    cinfo->marker->discarded_bytes += entropy->bitstate.bits_left / 8;
    entropy->bitstate.bits_left = 0;

    /* Advance past the RSTn marker */
    if (! (*cinfo->marker->read_restart_marker) (cinfo))
        return FALSE;
}

```

```

/* Re-initialize DC prediction to 0 */
for (ci = 0; ci < cinfo->blocks_in_scan; ci++)
    entropy->savd.last_dc_val[ci] = 0;
/* Re-init EOB run count, too */
entropy->savd.EOBRUN = 0;

/* Reset restart counter */
entropy->restarts_to_go = cinfo->restart_interval;

/* Reset out-of-data flag, unless read_restart_marker left us smack up
 * against a marker. In that case we will end up treating the next data
 * segment as empty, and we can avoid producing bogus output pixels by
 * leaving the flag set.
 */
if (cinfo->unread_marker == 0)
    entropy->pub.insufficient_data = FALSE;

return TRUE;
}

/*
 * Huffman MCU decoding.
 * Each of these routines decodes and returns one MCU's worth of
 * Huffman-compressed coefficients.
 * The coefficients are reordered from zigzag order into natural array order,
 * but are not dequantized.
 *
 * The i'th block of the MCU is stored into the block pointed to by
 * MCU_data[i]. WE ASSUME THIS AREA IS INITIALLY ZEROED BY THE CALLER.
 *
 * We return FALSE if data source requested suspension. In that case no
 * changes have been made to permanent state. (Exception: some output
 * coefficients may already have been assigned. This is harmless for
 * spectral selection, since we'll just re-assign them on the next call.
 * Successive approximation AC refinement has to be more careful, however.)
 */

/* MCU decoding for DC initial scan (either spectral selection,
 * or first pass of successive approximation).
 */
METHODDEF(boolean)
decode_mcu_DC_first (j_decompress_ptr cinfo, JBLOCKROW *MCU_data)
{
    phuff_entropy_ptr entropy = (phuff_entropy_ptr) cinfo->entropy;
    int Al = cinfo->Al;
    register int s, r;
    int blkcn, ci;
    JBLOCKROW block;
    BITREAD_STATE_VARS;
    savable_state state;
    d_derived_tbl * tbl;
    jpeg_component_info * compptr;

    /* Process restart marker if needed; may have to suspend */
    if (cinfo->restart_interval) {
        if (entropy->restarts_to_go == 0)
            if (! process_restart(cinfo))
                return FALSE;
    }

    /* If we've run out of data, just leave the MCU set to zeroes.
     * This way, we return uniform gray for the remainder of the segment.
     */
    if (! entropy->pub.insufficient_data) {
        /* Load up working state */
        BITREAD_LOAD_STATE(cinfo, entropy->bitstate);
        ASSIGN_STATE(state, entropy->savd);

        /* Outer loop handles each block in the MCU */
        for (blkcn = 0; blkcn < cinfo->blocks_in_MCU; blkcn++) {
            block = MCU_data[blkcn];
            ci = cinfo->MCU_membership[blkcn];
            compptr = cinfo->cur_comp_info[ci];
            tbl = entropy->derived_tbls[compptr->dc_tbl_no];

```

```

/* Decode a single block worth of coefficients */

/* Section F.2.2.1: decode the DC coefficient difference */
HUFF_DECODE(s, br_state, tbl, return FALSE, label1);
if (s) {
CHECK_BIT_BUFFER(br_state, s, return FALSE);
r = GET_BITS(s);
s = HUFF_EXTEND(r, s);
}

/* Convert DC difference to actual value, update last_dc_val */
s += state.last_dc_val[ci];
state.last_dc_val[ci] = s;
/* Scale and output the coefficient (assumes jpeg_natural_order[0]=0) */
(*block)[0] = (JCOEF) (s << A1);
}

/* Completed MCU, so update state */
BITREAD_SAVE_STATE(cinfo, entropy->bitstate);
ASSIGN_STATE(entropy->saved, state);
}

/* Account for restart interval (no-op if not using restarts) */
entropy->restarts_to_go--;

return TRUE;
}

/*
 * MCU decoding for AC initial scan (either spectral selection,
 * or first pass of successive approximation).
 */
METHODDEF(boolean)
decode_mcu_AC_first (j_decompress_ptr cinfo, JBLOCKROW *MCU_data)
{
    phuff_entropy_ptr entropy = (phuff_entropy_ptr) cinfo->entropy;
    int Se = cinfo->Se;
    int A1 = cinfo->A1;
    register int s, k, r;
    unsigned int EOBRUN;
    JBLOCKROW block;
    BITREAD_STATE_VARS;
    d_derived_tbl *tbl;

    /* Process restart marker if needed; may have to suspend */
    if (cinfo->restart_interval) {
        if (entropy->restarts_to_go == 0)
            if (! process_restart(cinfo))
                return FALSE;
    }

    /* If we've run out of data, just leave the MCU set to zeroes.
     * This way, we return uniform gray for the remainder of the segment.
     */
    if (! entropy->pub.insufficient_data) {

        /* Load up working state.
         * We can avoid loading/saving bitread state if in an EOB run.
         */
        EOBRUN = entropy->saved.EOBRUN; /* only part of saved state we need */

        /* There is always only one block per MCU */

        if (EOBRUN > 0) /* if it's a band of zeroes... */
            EOBRUN--; /* ...process it now (we do nothing) */
        else {
            BITREAD_LOAD_STATE(cinfo, entropy->bitstate);
            block = MCU_data[0];
            tbl = entropy->ac_derived_tbl;

            for (k = cinfo->Ss; k <= Se; k++) {
                HUFF_DECODE(s, br_state, tbl, return FALSE, label2);
                r = s >> 4;
                s &= 15;
                if (s) {
                    k += r;
                    CHECK_BIT_BUFFER(br_state, s, return FALSE);

```

```

    r = GET_BITS(s);
    s = HUFF_EXTEND(r, s);
    /* Scale and output coefficient in natural (dezigzagged) order */
    (*block)[jpeg_natural_order[k]] = (JCOEF) (s << A1);
} else {
    if (r == 15) { /* ZRL */
        k += 15; /* skip 15 zeroes in band */
    } else { /* EOB, run length is 2^r + appended bits */
        EOBRUN = 1 << r;
        if (r) { /* EOB, r > 0 */
            CHECK_BIT_BUFFER(br_state, r, return FALSE);
            r = GET_BITS(r);
            EOBRUN += r;
        }
        EOBRUN--; /* this band is processed at this moment */
        break; /* force end-of-band */
    }
}

BITREAD_SAVE_STATE(cinfo, entropy->bitstate);
}

/* Completed MCU, so update state */
entropy->saved.EOBRUN = EOBRUN; /* only part of saved state we need */
}

/* Account for restart interval (no-op if not using restarts) */
entropy->restarts_to_go--;

return TRUE;
}

/*
 * MCU decoding for DC successive approximation refinement scan.
 * Note: we assume such scans can be multi-component, although the spec
 * is not very clear on the point.
 */

METHODDEF(boolean)
decode_mcu_DC_refine (j_decompress_ptr cinfo, JBLOCKROW *MCU_data)
{
    phuff_entropy_ptr entropy = (phuff_entropy_ptr) cinfo->entropy;
    int pl = 1 << cinfo->A1; /* 1 in the bit position being coded */
    int blkcn;
    JBLOCKROW block;
    BITREAD_STATE_VARS;

    /* Process restart marker if needed; may have to suspend */
    if (cinfo->restart_interval) {
        if (entropy->restarts_to_go == 0)
            if (! process_restart(cinfo))
                return FALSE;
    }

    /* Not worth the cycles to check insufficient_data here,
     * since we will not change the data anyway if we read zeroes.
     */

    /* Load up working state */
    BITREAD_LOAD_STATE(cinfo, entropy->bitstate);

    /* Outer loop handles each block in the MCU */
    for (blkcn = 0; blkcn < cinfo->blocks_in_MCU; blkcn++) {
        block = MCU_data[blkcn];

        /* Encoded data is simply the next bit of the two's-complement DC value */
        CHECK_BIT_BUFFER(br_state, 1, return FALSE);
        if (GET_BITS(1))
            (*block)[0] |= pl;
        /* Note: since we use |=, repeating the assignment later is safe */
    }

    /* Completed MCU, so update state */
    BITREAD_SAVE_STATE(cinfo, entropy->bitstate);

    /* Account for restart interval (no-op if not using restarts) */
    entropy->restarts_to_go--;
}

```

```

    return TRUE;
}

/*
 * MCU decoding for AC successive approximation refinement scan.
 */

METHODDEF(boolean)
decode_mcu_AC_refine (j_decompress_ptr cinfo, JBLOCKROW *MCU_data)
{
    phuff_entropy_ptr entropy = (phuff_entropy_ptr) cinfo->entropy;
    int Se = cinfo->Se;
    int p1 = 1 << cinfo->A1; /* 1 in the bit position being coded */
    int m1 = (-1) << cinfo->A1; /* -1 in the bit position being coded */
    register int s, k, r;
    unsigned int EOBRUN;
    JBLOCKROW block;
    JCOEFPTR thiscoef;
    BITREAD_STATE_VARS;
    d_derived_tbl * tbl;
    int num_newnz;
    int newnz_pos[DCTSIZE2];

    /* Process restart marker if needed; may have to suspend */
    if (cinfo->restart_interval) {
        if (entropy->restarts_to_go == 0)
            if (! process_restart(cinfo))
                return FALSE;
    }

    /* If we've run out of data, don't modify the MCU.
     */
    if (! entropy->pub.insufficient_data) {
        /* Load up working state */
        BITREAD_LOAD_STATE(cinfo, entropy->bitstate);
        EOBRUN = entropy->saved.EOBRUN; /* only part of saved state we need */

        /* There is always only one block per MCU */
        block = MCU_data[0];
        tbl = entropy->ac_derived_tbl;

        /* If we are forced to suspend, we must undo the assignments to any newly
         * nonzero coefficients in the block, because otherwise we'd get confused
         * next time about which coefficients were already nonzero.
         * But we need not undo addition of bits to already-nonzero coefficients;
         * instead, we can test the current bit to see if we already did it.
         */
        num_newnz = 0;

        /* initialize coefficient loop counter to start of band */
        k = cinfo->Ss;

        if (EOBRUN == 0) {
            for (; k <= Se; k++) {
                HUFF_DECODE(s, br_state, tbl, goto undoit, label3);
                r = s >> 4;
                s &= 15;
                if (s) {
                    if (s != 1) /* size of new coef should always be 1 */
                        WARNMS(cinfo, JWRN_HUFF_BAD_CODE);
                    CHECK_BIT_BUFFER(br_state, 1, goto undoit);
                    if (GET_BITS(1))
                        s = p1; /* newly nonzero coef is positive */
                    else
                        s = m1; /* newly nonzero coef is negative */
                } else {
                    if (r != 15) {
                        EOBRUN = 1 << r; /* EOBr, run length is 2^r + appended bits */
                        if (r) {
                            CHECK_BIT_BUFFER(br_state, r, goto undoit);
                            r = GET_BITS(r);
                            EOBRUN += r;
                        }
                        break; /* rest of block is handled by EOB logic */
                    }
                }
                /* note s = 0 for processing ZRL */
            }
        }
    }
}

```

```

/* Advance over already-zero coefs and r still-zero coefs,
 * appending correction 1 to the nonzeros. A correction bit is 1
 * if the absolute value of the coefficient must be increased.
 */
do {
    thiscoef = *block + jpeg_natural_order[k];
    if (*thiscoef != 0) {
        CHECK_BIT_BUFFER(br_state, 1, goto undoit);
        if (GET_BITS(1)) {
            if ((*thiscoef & p1) == 0) { /* do nothing if already set it */
                if (*thiscoef >= 0)
                    *thiscoef += p1;
                else
                    *thiscoef += m1;
            }
        }
    } else {
        if (--r < 0)
            break; /* reached target zero coefficient */
    }
    k++;
} while (k <= Se);
if (s) {
    int pos = jpeg_natural_order[k];
    /* Output newly nonzero coefficient */
    (*block)[pos] = (JCOEF) s;
    /* Remember its position in case we have to suspend */
    newnz_pos[num_newnz++] = pos;
}
}

if (EOBRUN > 0) {
    /* Scan any remaining coefficient positions after the end-of-band
     * (the last newly nonzero coefficient, if any). Append a correction
     * bit to each already-nonzero coefficient. A correction bit is 1
     * if the absolute value of the coefficient must be increased.
     */
    for (; k <= Se; k++) {
        thiscoef = *block + jpeg_natural_order[k];
        if (*thiscoef != 0) {
            CHECK_BIT_BUFFER(br_state, 1, goto undoit);
            if (GET_BITS(1)) {
                if ((*thiscoef & p1) == 0) { /* do nothing if already changed it */
                    if (*thiscoef >= 0)
                        *thiscoef += p1;
                    else
                        *thiscoef += m1;
                }
            }
        }
    }
    /* Count one block completed in EOB run */
    EOBRUN--;
}

/* Completed MCU, so update state */
BITREAD_SAVE_STATE(cinfo, entropy->bitstate);
entropy->saved.EOBRUN = EOBRUN; /* only part of saved state we need */
}

/* Account for restart interval (no-op if not using restarts) */
entropy->restarts_to_go--;

return TRUE;

undoit:
/* Re-zero any output coefficients that we made newly nonzero */
while (num_newnz > 0)
    (*block)[newnz_pos[--num_newnz]] = 0;

return FALSE;
}

/*
 * Module initialization routine for progressive Huffman entropy decoding.
 */
GLOBAL(void)

```

```

jinit_phuff_decoder (j_decompress_ptr cinfo)
{
    phuff_entropy_ptr entropy;
    int *coef_bit_ptr;
    int ci, i;

    entropy = (phuff_entropy_ptr)
        (*cinfo->mem->alloc_small) ((j_common_ptr) cinfo, JPOOL_IMAGE,
            SIZEOF(phuff_entropy_decoder));
    cinfo->entropy = (struct jpeg_entropy_decoder *) entropy;
    entropy->pub.start_pass = start_pass_phuff_decoder;

    /* Mark derived tables unallocated */
    for (i = 0; i < NUM_HUFF_TBLS; i++) {
        entropy->derived_tbls[i] = NULL;
    }

    /* Create progression status table */
    cinfo->coef_bits = (int (*)[DCTSIZE2])
        (*cinfo->mem->alloc_small) ((j_common_ptr) cinfo, JPOOL_IMAGE,
            cinfo->num_components*DCTSIZE2*SIZEOF(int));
    coef_bit_ptr = & cinfo->coef_bits[0][0];
    for (ci = 0; ci < cinfo->num_components; ci++)
        for (i = 0; i < DCTSIZE2; i++)
            *coef_bit_ptr++ = -1;
}

#endif /* D_PROGRESSIVE_SUPPORTED */

```

```

/*
 * jdpostct.c
 *
 * Copyright (C) 1994-1996, Thomas G. Lane.
 * This file is part of the Independent JPEG Group's software.
 * For conditions of distribution and use, see the accompanying README file.
 *
 * This file contains the decompression postprocessing controller.
 * This controller manages the upsampling, color conversion, and color
 * quantization/reduction steps; specifically, it controls the buffering
 * between upsample/color conversion and color quantization/reduction.
 *
 * If no color quantization/reduction is required, then this module has no
 * work to do, and it just hands off to the upsample/color conversion code.
 * An integrated upsample/convert/quantize process would replace this module
 * entirely.
 */

#define JPEG_INTERNALS
#include "jinclude.h"
#include "jpeglib.h"

/* Private buffer controller object */

typedef struct {
  struct jpeg_d_post_controller pub; /* public fields */

  /* Color quantization source buffer: this holds output data from
   * the upsample/color conversion step to be passed to the quantizer.
   * For two-pass color quantization, we need a full-image buffer;
   * for one-pass operation, a strip buffer is sufficient.
   */
  JSAMPARRAY whole_image; /* virtual array, or NULL if one-pass */
  JSAMPARRAY buffer; /* strip buffer, or current strip of virtual */
  JDIMENSION strip_height; /* buffer size in rows */
  /* for two-pass mode only: */
  JDIMENSION starting_row; /* row # of first row in current strip */
  JDIMENSION next_row; /* index of next row to fill/empty in strip */
} my_post_controller;

typedef my_post_controller * my_post_ptr;

/* Forward declarations */
METHODDEF(void) post_process_1pass
    JPP((j_decompress_ptr cinfo,
         JSAMPIMAGE input_buf, JDIMENSION *in_row_group_ctr,
         JDIMENSION in_row_groups_avail,
         JSAMPARRAY output_buf, JDIMENSION *out_row_ctr,
         JDIMENSION out_rows_avail));
#ifdef QUANT_2PASS_SUPPORTED
METHODDEF(void) post_process_prepass
    JPP((j_decompress_ptr cinfo,
         JSAMPIMAGE input_buf, JDIMENSION *in_row_group_ctr,
         JDIMENSION in_row_groups_avail,
         JSAMPARRAY output_buf, JDIMENSION *out_row_ctr,
         JDIMENSION out_rows_avail));
METHODDEF(void) post_process_2pass
    JPP((j_decompress_ptr cinfo,
         JSAMPIMAGE input_buf, JDIMENSION *in_row_group_ctr,
         JDIMENSION in_row_groups_avail,
         JSAMPARRAY output_buf, JDIMENSION *out_row_ctr,
         JDIMENSION out_rows_avail));
#endif

/*
 * Initialize for a processing pass.
 */

METHODDEF(void)
start_pass_dpost (j_decompress_ptr cinfo, J_BUF_MODE pass_mode)
{
  my_post_ptr post = (my_post_ptr) cinfo->post;

  switch (pass_mode) {
    case JBUF_PASS_THRU:
      if (cinfo->quantize_colors) {
        /* Single-pass processing with color quantization. */

```



```

    post->pub.post_process_data = post_process_lpass;
    /* We could be doing buffered-image output before starting 2-pass
     * color quantization; in that case, jinit_d_post_controller would not
     * allocate a strip buffer. Use the virtual-array buffer as workspace.
     */
    if (post->buffer == NULL) {
        post->buffer = (*cinfo->mem->access_virt_sarray)
            ((j_common_ptr) cinfo, post->whole_image,
             (JDIMENSION) 0, post->strip_height, TRUE);
    }
} else {
    /* For single-pass processing without color quantization,
     * I have no work to do; just call the upsampler directly.
     */
    post->pub.post_process_data = cinfo->upsample->upsample;
}
break;
#endif QUANT_2PASS_SUPPORTED
case JBUF_SAVE_AND_PASS:
    /* First pass of 2-pass quantization */
    if (post->whole_image == NULL)
        ERREXIT(cinfo, JERR_BAD_BUFFER_MODE);
    post->pub.post_process_data = post_process_prepass;
    break;
case JBUF_CRANK_DEST:
    /* Second pass of 2-pass quantization */
    if (post->whole_image == NULL)
        ERREXIT(cinfo, JERR_BAD_BUFFER_MODE);
    post->pub.post_process_data = post_process_2pass;
    break;
#endif /* QUANT_2PASS_SUPPORTED */
default:
    ERREXIT(cinfo, JERR_BAD_BUFFER_MODE);
    break;
}
post->starting_row = post->next_row = 0;
}

/*
 * Process some data in the one-pass (strip buffer) case.
 * This is used for color precision reduction as well as one-pass quantization.
 */
METHODDEF(void)
post_process_lpass (j_decompress_ptr cinfo,
                    JSAMPIMAGE input_buf, JDIMENSION *in_row_group_ctr,
                    JDIMENSION in_row_groups_avail,
                    JSAMPARRAY output_buf, JDIMENSION *out_row_ctr,
                    JDIMENSION out_rows_avail)
{
    my_post_ptr post = (my_post_ptr) cinfo->post;
    JDIMENSION num_rows, max_rows;

    /* Fill the buffer, but not more than what we can dump out in one go. */
    /* Note we rely on the upsampler to detect bottom of image. */
    max_rows = out_rows_avail - *out_row_ctr;
    if (max_rows > post->strip_height)
        max_rows = post->strip_height;
    num_rows = 0;
    (*cinfo->upsample->upsample) (cinfo,
                                input_buf, in_row_group_ctr, in_row_groups_avail,
                                post->buffer, &num_rows, max_rows);
    /* Quantize and emit data. */
    (*cinfo->cquantize->color_quantize) (cinfo,
                                        post->buffer, output_buf + *out_row_ctr, (int) num_rows);
    *out_row_ctr += num_rows;
}

#ifdef QUANT_2PASS_SUPPORTED
/*
 * Process some data in the first pass of 2-pass quantization.
 */
METHODDEF(void)
post_process_prepass (j_decompress_ptr cinfo,
                     JSAMPIMAGE input_buf, JDIMENSION *in_row_group_ctr,
                     JDIMENSION in_row_groups_avail,

```

```

        JSAMPARRAY output_buf, JDIMENSION *out_row_ctr,
        JDIMENSION out_rows_avail)
{
    my_post_ptr post = (my_post_ptr) cinfo->post;
    JDIMENSION old_next_row, num_rows;

    /* Reposition virtual buffer if at start of strip. */
    if (post->next_row == 0) {
        post->buffer = (*cinfo->mem->access_virt_sarray)
            ((j_common_ptr) cinfo, post->whole_image,
            post->starting_row, post->strip_height, TRUE);
    }

    /* Upsample some data (up to a strip height's worth). */
    old_next_row = post->next_row;
    (*cinfo->upsample->upsample) (cinfo,
        input_buf, in_row_group_ctr, in_row_groups_avail,
        post->buffer, &post->next_row, post->strip_height);

    /* Allow quantizer to scan new data. No data is emitted, */
    /* but we advance out_row_ctr so outer loop can tell when we're done. */
    if (post->next_row > old_next_row) {
        num_rows = post->next_row - old_next_row;
        (*cinfo->cquantize->color_quantize) (cinfo, post->buffer + old_next_row,
            (JSAMPARRAY) NULL, (int) num_rows);
        *out_row_ctr += num_rows;
    }

    /* Advance if we filled the strip. */
    if (post->next_row >= post->strip_height) {
        post->starting_row += post->strip_height;
        post->next_row = 0;
    }

    /* Process some data in the second pass of 2-pass quantization. */

METHODDEF(void)
post_process_2pass (j_decompress_ptr cinfo,
    JSAMPIMAGE input_buf, JDIMENSION *in_row_group_ctr,
    JDIMENSION in_row_groups_avail,
    JSAMPARRAY output_buf, JDIMENSION *out_row_ctr,
    JDIMENSION out_rows_avail)
{
    my_post_ptr post = (my_post_ptr) cinfo->post;
    JDIMENSION num_rows, max_rows;

    /* Reposition virtual buffer if at start of strip. */
    if (post->next_row == 0) {
        post->buffer = (*cinfo->mem->access_virt_sarray)
            ((j_common_ptr) cinfo, post->whole_image,
            post->starting_row, post->strip_height, FALSE);
    }

    /* Determine number of rows to emit. */
    num_rows = post->strip_height - post->next_row; /* available in strip */
    max_rows = out_rows_avail - *out_row_ctr; /* available in output area */
    if (num_rows > max_rows)
        num_rows = max_rows;
    /* We have to check bottom of image here, can't depend on upsampler. */
    max_rows = cinfo->output_height - post->starting_row;
    if (num_rows > max_rows)
        num_rows = max_rows;

    /* Quantize and emit data. */
    (*cinfo->cquantize->color_quantize) (cinfo,
        post->buffer + post->next_row, output_buf + *out_row_ctr,
        (int) num_rows);
    *out_row_ctr += num_rows;

    /* Advance if we filled the strip. */
    post->next_row += num_rows;
    if (post->next_row >= post->strip_height) {
        post->starting_row += post->strip_height;
        post->next_row = 0;
    }
}

```

```
#endif /* QUANT_2PASS_SUPPORTED */
```

```
/*  
 * Initialize postprocessing controller.  
 */
```

```
GLOBAL(void)
```

```
jinit_d_post_controller (j_decompress_ptr cinfo, boolean need_full_buffer)
```

```
{  
    my_post_ptr post;  
  
    post = (my_post_ptr)  
        (*cinfo->mem->alloc_small) ((j_common_ptr) cinfo, JPOOL_IMAGE,  
        SIZEOF(my_post_controller));  
    cinfo->post = (struct jpeg_d_post_controller *) post;  
    post->pub.start_pass = start_pass_dpost;  
    post->whole_image = NULL; /* flag for no virtual arrays */  
    post->buffer = NULL; /* flag for no strip buffer */  
  
    /* Create the quantization buffer, if needed */  
    if (cinfo->quantize_colors) {  
        /* The buffer strip height is max_v_samp_factor, which is typically  
        * an efficient number of rows for upsampling to return.  
        * (In the presence of output rescaling, we might want to be smarter?)  
        */  
        post->strip_height = (JDIMENSION) cinfo->max_v_samp_factor;  
        if (need_full_buffer) {  
            /* Two-pass color quantization: need full-image storage. */  
            /* We round up the number of rows to a multiple of the strip height. */  
#ifdef QUANT_2PASS_SUPPORTED  
            post->whole_image = (*cinfo->mem->request_virt_sarray)  
                ((j_common_ptr) cinfo, JPOOL_IMAGE, FALSE,  
                cinfo->output_width * cinfo->out_color_components,  
                (JDIMENSION) jround_up((long) cinfo->output_height,  
                (long) post->strip_height),  
                (long) post->strip_height);  
#else  
            ERREXIT(cinfo, JERR_BAD_BUFFER_MODE);  
#endif /* QUANT_2PASS_SUPPORTED */  
        } else {  
            /* One-pass color quantization: just make a strip buffer. */  
            post->buffer = (*cinfo->mem->alloc_sarray)  
                ((j_common_ptr) cinfo, JPOOL_IMAGE,  
                cinfo->output_width * cinfo->out_color_components,  
                post->strip_height);  
        }  
    }  
}
```

```

/*
 * jdsample.c
 *
 * Copyright (C) 1991-1996, Thomas G. Lane.
 * This file is part of the Independent JPEG Group's software.
 * For conditions of distribution and use, see the accompanying README file.
 *
 * This file contains upsampling routines.
 *
 * Upsampling input data is counted in "row groups".  A row group
 * is defined to be (v_samp_factor * DCT_scaled_size / min_DCT_scaled_size)
 * sample rows of each component.  Upsampling will normally produce
 * max_v_samp_factor pixel rows from each row group (but this could vary
 * if the upsampler is applying a scale factor of its own).
 *
 * An excellent reference for image resampling is
 * Digital Image Warping, George Wolberg, 1990.
 * Pub. by IEEE Computer Society Press, Los Alamitos, CA. ISBN 0-8186-8944-7.
 */

```

```

#define JPEG_INTERNALS
#include "jinclude.h"
#include "jpeglib.h"

```

```

/* Pointer to routine to upsample a single component */
typedef JMETHODOF(void, upsampler_ptr,
  (j_decompress_ptr cinfo, jpeg_component_info * comp_ptr,
   JSAMPARRAY input_data, JSAMPARRAY * output_data_ptr));

```

```

/* Private subobject */

```

```

typedef struct {
  struct jpeg_upsampler pub; /* public fields */

  /* Color conversion buffer.  When using separate upsampling and color
   * conversion steps, this buffer holds one upsampled row group until it
   * has been color converted and output.
   * Note: we do not allocate any storage for component(s) which are full-size,
   * ie do not need rescaling.  The corresponding entry of color_buf[] is
   * simply set to point to the input data array, thereby avoiding copying.
   */
  JSAMPARRAY color_buf[MAX_COMPONENTS];

  /* Per-component upsampling method pointers */
  upsampler_ptr methods[MAX_COMPONENTS];

  int next_row_out; /* counts rows emitted from color_buf */
  JDIMENSION rows_to_go; /* counts rows remaining in image */

  /* Height of an input row group for each component. */
  int rowgroup_height[MAX_COMPONENTS];

  /* These arrays save pixel expansion factors so that int_expand need not
   * recompute them each time.  They are unused for other upsampling methods.
   */
  UINT8 h_expand[MAX_COMPONENTS];
  UINT8 v_expand[MAX_COMPONENTS];
} my_upsampler;

```

```

typedef my_upsampler * my_upsample_ptr;

```

```

/*
 * Initialize for an upsampling pass.
 */

```

```

METHODDEF(void)
start_pass_upsample (j_decompress_ptr cinfo)
{
  my_upsample_ptr upsample = (my_upsample_ptr) cinfo->upsample;

  /* Mark the conversion buffer empty */
  upsample->next_row_out = cinfo->max_v_samp_factor;
  /* Initialize total-height counter for detecting bottom of image */
  upsample->rows_to_go = cinfo->output_height;
}

```

```

/*

```

```

* Control routine to do upsampling (and color conversion).
*
* In this version we upsample each component independently.
* We upsample one row group into the conversion buffer, then apply
* color conversion a row at a time.
*/

```

```

METHODDEF(void)

```

```

sep_upsample (j_decompress_ptr cinfo,
              JSAMPIMAGE input_buf, JDIMENSION *in_row_group_ctr,
              JDIMENSION in_row_groups_avail,
              JSAMPARRAY output_buf, JDIMENSION *out_row_ctr,
              JDIMENSION out_rows_avail)
{
    my_upsample_ptr upsample = (my_upsample_ptr) cinfo->upsample;
    int ci;
    jpeg_component_info * compptr;
    JDIMENSION num_rows;

    /* Fill the conversion buffer, if it's empty */
    if (upsample->next_row_out >= cinfo->max_v_samp_factor) {
        for (ci = 0, compptr = cinfo->comp_info; ci < cinfo->num_components;
            ci++, compptr++) {
            /* Invoke per-component upsample method. Notice we pass a POINTER
             * to color_buf[ci], so that fullsize_upsample can change it.
             */
            (*upsample->methods[ci]) (cinfo, compptr,
                input_buf[ci] + (*in_row_group_ctr * upsample->rowgroup_height[ci]),
                upsample->color_buf + ci);
        }
        upsample->next_row_out = 0;

        /* Color-convert and emit rows */

        /* How many we have in the buffer: */
        num_rows = (JDIMENSION) (cinfo->max_v_samp_factor - upsample->next_row_out);
        /* Not more than the distance to the end of the image. Need this test
         * in case the image height is not a multiple of max_v_samp_factor:
         */
        if (num_rows > upsample->rows_to_go)
            num_rows = upsample->rows_to_go;
        /* And not more than what the client can accept: */
        num_rows -= *out_row_ctr;
        if (num_rows > out_rows_avail)
            num_rows = out_rows_avail;

        (*cinfo->cconvert->color_convert) (cinfo, upsample->color_buf,
            (JDIMENSION) upsample->next_row_out,
            output_buf + *out_row_ctr,
            (int) num_rows);

        /* Adjust counts */
        *out_row_ctr += num_rows;
        upsample->rows_to_go -= num_rows;
        upsample->next_row_out += num_rows;
        /* When the buffer is emptied, declare this input row group consumed */
        if (upsample->next_row_out >= cinfo->max_v_samp_factor)
            (*in_row_group_ctr)++;
    }
}

```

```

/*
 * These are the routines invoked by sep_upsample to upsample pixel values
 * of a single component. One row group is processed per call.
 */

```

```

/*
 * For full-size components, we just make color_buf[ci] point at the
 * input buffer, and thus avoid copying any data. Note that this is
 * safe only because sep_upsample doesn't declare the input row group
 * "consumed" until we are done color converting and emitting it.
 */

```

```

METHODDEF(void)

```

```

fullsize_upsample (j_decompress_ptr cinfo, jpeg_component_info * compptr,
                  JSAMPARRAY input_data, JSAMPARRAY * output_data_ptr)
{
    *output_data_ptr = input_data;
}

```

```

)

/*
 * This is a no-op version used for "uninteresting" components.
 * These components will not be referenced by color conversion.
 */

METHODDEF(void)
noop_upsample (j_decompress_ptr cinfo, jpeg_component_info * compptr,
               JSAMPARRAY input_data, JSAMPARRAY * output_data_ptr)
{
    *output_data_ptr = NULL; /* safety check */
}

/*
 * This version handles any integral sampling ratios.
 * This is not used for typical JPEG files, so it need not be fast.
 * Nor, for that matter, is it particularly accurate: the algorithm is
 * simple replication of the input pixel onto the corresponding output
 * pixels. The hi-falutin sampling literature refers to this as a
 * "box filter". A box filter tends to introduce visible artifacts,
 * so if you are actually going to use 3:1 or 4:1 sampling ratios
 * you would be well advised to improve this code.
 */

METHODDEF(void)
int_upsample (j_decompress_ptr cinfo, jpeg_component_info * compptr,
              JSAMPARRAY input_data, JSAMPARRAY * output_data_ptr)
{
    my_upsample_ptr upsample = (my_upsample_ptr) cinfo->upsample;
    JSAMPARRAY output_data = *output_data_ptr;
    register JSAMPROW inptr, outptr;
    register JSAMPLE invalue;
    register int h;
    JSAMPROW outend;
    int h_expand, v_expand;
    int inrow, outrow;

    h_expand = upsample->h_expand[compptr->component_index];
    v_expand = upsample->v_expand[compptr->component_index];

    inrow = outrow = 0;
    while (outrow < cinfo->max_v_samp_factor) {
        /* Generate one output row with proper horizontal expansion */
        inptr = input_data[inrow];
        outptr = output_data[outrow];
        outend = outptr + cinfo->output_width;
        while (outptr < outend) {
            invalue = *inptr++; /* don't need GETJSAMPLE() here */
            for (h = h_expand; h > 0; h--) {
                *outptr++ = invalue;
            }
        }
        /* Generate any additional output rows by duplicating the first one */
        if (v_expand > 1) {
            jcopy_sample_rows(output_data, outrow, output_data, outrow+1,
                              v_expand-1, cinfo->output_width);
        }
        inrow++;
        outrow += v_expand;
    }
}

/*
 * Fast processing for the common case of 2:1 horizontal and 1:1 vertical.
 * It's still a box filter.
 */

METHODDEF(void)
h2v1_upsample (j_decompress_ptr cinfo, jpeg_component_info * compptr,
               JSAMPARRAY input_data, JSAMPARRAY * output_data_ptr)
{
    JSAMPARRAY output_data = *output_data_ptr;
    register JSAMPROW inptr, outptr;
    register JSAMPLE invalue;
    JSAMPROW outend;
    int inrow;

```

```

for (inrow = 0; inrow < cinfo->max_v_samp_factor; inrow++) {
    inptr = input_data[inrow];
    outptr = output_data[inrow];
    outend = outptr + cinfo->output_width;
    while (outptr < outend) {
        invalue = *inptr++; /* don't need GETJSAMPLE() here */
        *outptr++ = invalue;
        *outptr++ = invalue;
    }
}

/*
 * Fast processing for the common case of 2:1 horizontal and 2:1 vertical.
 * It's still a box filter.
 */

METHODDEF(void)
h2v2_upsample (j_decompress_ptr cinfo, jpeg_component_info * compptr,
               JSAMPARRAY input_data, JSAMPARRAY * output_data_ptr)
{
    JSAMPARRAY output_data = *output_data_ptr;
    register JSAMPROW inptr, outptr;
    register JSAMPLE invalue;
    JSAMPROW outend;
    int inrow, outrow;

    inrow = outrow = 0;
    while (outrow < cinfo->max_v_samp_factor) {
        inptr = input_data[inrow];
        outptr = output_data[outrow];
        outend = outptr + cinfo->output_width;
        while (outptr < outend) {
            invalue = *inptr++; /* don't need GETJSAMPLE() here */
            *outptr++ = invalue;
            *outptr++ = invalue;
        }
        jcopy_sample_rows(output_data, outrow, output_data, outrow+1,
                          1, cinfo->output_width);
        inrow++;
        outrow += 2;
    }
}

/*
 * Fancy processing for the common case of 2:1 horizontal and 1:1 vertical.
 *
 * The upsampling algorithm is linear interpolation between pixel centers,
 * also known as a "triangle filter". This is a good compromise between
 * speed and visual quality. The centers of the output pixels are 1/4 and 3/4
 * of the way between input pixel centers.
 *
 * A note about the "bias" calculations: when rounding fractional values to
 * integer, we do not want to always round 0.5 up to the next integer.
 * If we did that, we'd introduce a noticeable bias towards larger values.
 * Instead, this code is arranged so that 0.5 will be rounded up or down at
 * alternate pixel locations (a simple ordered dither pattern).
 */

METHODDEF(void)
h2v1_fancy_upsample (j_decompress_ptr cinfo, jpeg_component_info * compptr,
                    JSAMPARRAY input_data, JSAMPARRAY * output_data_ptr)
{
    JSAMPARRAY output_data = *output_data_ptr;
    register JSAMPROW inptr, outptr;
    register int invalue;
    register JDIMENSION colctr;
    int inrow;

    for (inrow = 0; inrow < cinfo->max_v_samp_factor; inrow++) {
        inptr = input_data[inrow];
        outptr = output_data[inrow];
        /* Special case for first column */
        invalue = GETJSAMPLE(*inptr++);
        *outptr++ = (JSAMPLE) invalue;
        *outptr++ = (JSAMPLE) ((invalue * 3 + GETJSAMPLE(*inptr) + 2) >> 2);
    }
}

```

```

    for (colctr = compptr->downsampled_width - 2; colctr > 0; colctr--) {
        /* General case: 3/4 * nearer pixel + 1/4 * further pixel */
        invalue = GETJSAMPLE(*inptr++) * 3;
        *outptr++ = (JSAMPLE) ((invalue + GETJSAMPLE(inptr[-2]) + 1) >> 2);
        *outptr++ = (JSAMPLE) ((invalue + GETJSAMPLE(*inptr) + 2) >> 2);
    }

    /* Special case for last column */
    invalue = GETJSAMPLE(*inptr);
    *outptr++ = (JSAMPLE) ((invalue * 3 + GETJSAMPLE(inptr[-1]) + 1) >> 2);
    *outptr++ = (JSAMPLE) invalue;
}

/*
 * Fancy processing for the common case of 2:1 horizontal and 2:1 vertical.
 * Again a triangle filter; see comments for h2v1 case, above.
 *
 * It is OK for us to reference the adjacent input rows because we demanded
 * context from the main buffer controller (see initialization code).
 */

METHODDEF(void)
h2v2_fancy_upsample (j_decompress_ptr cinfo, jpeg_component_info * compptr,
                    JSAMPARRAY input_data, JSAMPARRAY * output_data_ptr)
{
    JSAMPARRAY output_data = *output_data_ptr;
    register JSAMPROW inptr0, inptr1, outptr;
    #if BITS_IN_JSAMPLE == 8
        register int thiscolsum, lastcolsum, nextcolsum;
    #else
        register INT32 thiscolsum, lastcolsum, nextcolsum;
    #endif
    register JDIMENSION colctr;
    int inrow, outrow, v;

    inrow = outrow = 0;
    while (outrow < cinfo->max_v_samp_factor) {
        for (v = 0; v < 2; v++) {
            /* inptr0 points to nearest input row, inptr1 points to next nearest */
            inptr0 = input_data[inrow];
            if (v == 0) /* next nearest is row above */
                inptr1 = input_data[inrow-1];
            else /* next nearest is row below */
                inptr1 = input_data[inrow+1];
            outptr = output_data[outrow++];

            /* Special case for first column */
            thiscolsum = GETJSAMPLE(*inptr0++) * 3 + GETJSAMPLE(*inptr1++);
            nextcolsum = GETJSAMPLE(*inptr0++) * 3 + GETJSAMPLE(*inptr1++);
            *outptr++ = (JSAMPLE) ((thiscolsum * 4 + 8) >> 4);
            *outptr++ = (JSAMPLE) ((thiscolsum * 3 + nextcolsum + 7) >> 4);
            lastcolsum = thiscolsum; thiscolsum = nextcolsum;

            for (colctr = compptr->downsampled_width - 2; colctr > 0; colctr--) {
                /* General case: 3/4 * nearer pixel + 1/4 * further pixel in each */
                /* dimension, thus 9/16, 3/16, 3/16, 1/16 overall */
                nextcolsum = GETJSAMPLE(*inptr0++) * 3 + GETJSAMPLE(*inptr1++);
                *outptr++ = (JSAMPLE) ((thiscolsum * 3 + lastcolsum + 8) >> 4);
                *outptr++ = (JSAMPLE) ((thiscolsum * 3 + nextcolsum + 7) >> 4);
                lastcolsum = thiscolsum; thiscolsum = nextcolsum;
            }

            /* Special case for last column */
            *outptr++ = (JSAMPLE) ((thiscolsum * 3 + lastcolsum + 8) >> 4);
            *outptr++ = (JSAMPLE) ((thiscolsum * 4 + 7) >> 4);
        }
        inrow++;
    }
}

/*
 * Module initialization routine for upsampling.
 */

GLOBAL(void)
jinit_upsampler (j_decompress_ptr cinfo)
{

```



```

my_upsample_ptr upsample;
int ci;
jpeg_component_info * comp_ptr;
boolean need_buffer, do_fancy;
int h_in_group, v_in_group, h_out_group, v_out_group;

upsample = (my_upsample_ptr)
    (*cinfo->mem->alloc_small) ((j_common_ptr) cinfo, JPOOL_IMAGE,
        SIZEOF(my_upsampler));
cinfo->upsample = (struct jpeg_upsampler *) upsample;
upsample->pub.start_pass = start_pass_upsample;
upsample->pub.upsample = sep_upsample;
upsample->pub.need_context_rows = FALSE; /* until we find out differently */

if (cinfo->CCIR601_sampling) /* this isn't supported */
    ERREXIT(cinfo, JERR_CCIR601_NOTIMPL);

/* jdmainct.c doesn't support context rows when min_DCT_scaled_size = 1,
 * so don't ask for it.
 */
do_fancy = cinfo->do_fancy_upsampling && cinfo->min_DCT_scaled_size > 1;

/* Verify we can handle the sampling factors, select per-component methods,
 * and create storage as needed.
 */
for (ci = 0, comp_ptr = cinfo->comp_info; ci < cinfo->num_components;
    ci++, comp_ptr++) {
    /* Compute size of an "input group" after IDCT scaling. This many samples
     * are to be converted to max_h_samp_factor * max_v_samp_factor pixels.
     */
    h_in_group = (comp_ptr->h_samp_factor * comp_ptr->DCT_scaled_size) /
        cinfo->min_DCT_scaled_size;
    v_in_group = (comp_ptr->v_samp_factor * comp_ptr->DCT_scaled_size) /
        cinfo->min_DCT_scaled_size;
    h_out_group = cinfo->max_h_samp_factor;
    v_out_group = cinfo->max_v_samp_factor;
    upsample->rowgroup_height[ci] = v_in_group; /* save for use later */
    need_buffer = TRUE;
    if (! comp_ptr->component_needed) {
        /* Don't bother to upsample an uninteresting component. */
        upsample->methods[ci] = noop_upsample;
        need_buffer = FALSE;
    } else if (h_in_group == h_out_group && v_in_group == v_out_group) {
        /* Fullsize components can be processed without any work. */
        upsample->methods[ci] = fullsize_upsample;
        need_buffer = FALSE;
    } else if (h_in_group * 2 == h_out_group &&
        v_in_group == v_out_group) {
        /* Special cases for 2h1v upsampling */
        if (do_fancy && comp_ptr->downsampled_width > 2)
            upsample->methods[ci] = h2v1_fancy_upsample;
        else
            upsample->methods[ci] = h2v1_upsample;
    } else if (h_in_group * 2 == h_out_group &&
        v_in_group * 2 == v_out_group) {
        /* Special cases for 2h2v upsampling */
        if (do_fancy && comp_ptr->downsampled_width > 2) {
            upsample->methods[ci] = h2v2_fancy_upsample;
            upsample->pub.need_context_rows = TRUE;
        } else
            upsample->methods[ci] = h2v2_upsample;
    } else if ((h_out_group % h_in_group) == 0 &&
        (v_out_group % v_in_group) == 0) {
        /* Generic integral-factors upsampling method */
        upsample->methods[ci] = int_upsample;
        upsample->h_expand[ci] = (UINT8) (h_out_group / h_in_group);
        upsample->v_expand[ci] = (UINT8) (v_out_group / v_in_group);
    } else
        ERREXIT(cinfo, JERR_FRACT_SAMPLE_NOTIMPL);
    if (need_buffer) {
        upsample->color_buf[ci] = (*cinfo->mem->alloc_sarray)
            ((j_common_ptr) cinfo, JPOOL_IMAGE,
                (JDIMENSION) jround_up((long) cinfo->output_width,
                    (long) cinfo->max_h_samp_factor),
                (JDIMENSION) cinfo->max_v_samp_factor);
    }
}
}

```

```

/*
 * jdtrans.c
 *
 * Copyright (C) 1995-1997, Thomas G. Lane.
 * This file is part of the Independent JPEG Group's software.
 * For conditions of distribution and use, see the accompanying README file.
 *
 * This file contains library routines for transcoding decompression,
 * that is, reading raw DCT coefficient arrays from an input JPEG file.
 * The routines in jdapimin.c will also be needed by a transcoder.
 */

#define JPEG_INTERNALS
#include "jinclude.h"
#include "jpeglib.h"

/* Forward declarations */
LOCAL(void) transdecode_master_selection JPP((j_decompress_ptr cinfo));

/*
 * Read the coefficient arrays from a JPEG file.
 * jpeg_read_header must be completed before calling this.
 *
 * The entire image is read into a set of virtual coefficient-block arrays,
 * one per component. The return value is a pointer to the array of
 * virtual-array descriptors. These can be manipulated directly via the
 * JPEG memory manager, or handed off to jpeg_write_coefficients().
 * To release the memory occupied by the virtual arrays, call
 * jpeg_finish_decompress() when done with the data.
 *
 * An alternative usage is to simply obtain access to the coefficient arrays
 * during a buffered-image-mode decompression operation. This is allowed
 * after any jpeg_finish_output() call. The arrays can be accessed until
 * jpeg_finish_decompress() is called. (Note that any call to the library
 * may reposition the arrays, so don't rely on access_virt_barray() results
 * to stay valid across library calls.)
 *
 * Returns NULL if suspended. This case need be checked only if
 * a suspending data source is used.
 */

GLOBAL(jvirt_barray_ptr *)
jpeg_read_coefficients(j_decompress_ptr cinfo)
{
  if (cinfo->global_state == DSTATE_READY) {
    /* First call: initialize active modules */
    transdecode_master_selection(cinfo);
    cinfo->global_state = DSTATE_RDCOEFS;
  }
  if (cinfo->global_state == DSTATE_RDCOEFS) {
    /* Absorb whole file into the coef buffer */
    for (;;) {
      int retcode;
      /* Call progress monitor hook if present */
      if (cinfo->progress != NULL)
        (*cinfo->progress->progress_monitor) ((j_common_ptr) cinfo);
      /* Absorb some more input */
      retcode = (*cinfo->inputctl->consume_input) (cinfo);
      if (retcode == JPEG_SUSPENDED)
        return NULL;
      if (retcode == JPEG_REACHED_EOI)
        break;
      /* Advance progress counter if appropriate */
      if (cinfo->progress != NULL &&
          (retcode == JPEG_ROW_COMPLETED || retcode == JPEG_REACHED_SOS)) {
        if (++cinfo->progress->pass_counter >= cinfo->progress->pass_limit) {
          /* startup underestimated number of scans; ratchet up one scan */
          cinfo->progress->pass_limit += (long) cinfo->total_iMCU_rows;
        }
      }
    }
    /* Set state so that jpeg_finish_decompress does the right thing */
    cinfo->global_state = DSTATE_STOPPING;
  }
  /* At this point we should be in state DSTATE_STOPPING if being used
   * standalone, or in state DSTATE_BUFIMAGE if being invoked to get access
   * to the coefficients during a full buffered-image-mode decompression.
   */
}

```

```

if ((cinfo->global_state == STATE_STOPPING ||
    cinfo->global_state == STATE_BUFIMAGE) && cinfo->buffered_image) {
    return cinfo->coef->coef_arrays;
}
/* Oops, improper usage */
ERREXIT1(cinfo, JERR_BAD_STATE, cinfo->global_state);
return NULL; /* keep compiler happy */
}

/*
 * Master selection of decompression modules for transcoding.
 * This substitutes for jdmaster.c's initialization of the full decompressor.
 */

LOCAL(void)
transdecode_master_selection (j_decompress_ptr cinfo)
{
    /* This is effectively a buffered-image operation. */
    cinfo->buffered_image = TRUE;

    /* Entropy decoding: either Huffman or arithmetic coding. */
    if (cinfo->arith_code) {
        ERREXIT(cinfo, JERR_ARITH_NOTIMPL);
    } else {
        if (cinfo->progressive_mode) {
#ifdef D_PROGRESSIVE_SUPPORTED
            jinit_phuff_decoder(cinfo);
#else
            ERREXIT(cinfo, JERR_NOT_COMPILED);
#endif
        } else {
            jinit_huff_decoder(cinfo);
        }

        /* Always get a full-image coefficient buffer. */
        jinit_d_coef_controller(cinfo, TRUE);

        /* We can now tell the memory manager to allocate virtual arrays. */
        (*cinfo->mem->realize_virt_arrays) ((j_common_ptr) cinfo);

        /* Initialize input side of decompressor to consume first scan. */
        (*cinfo->inputctl->start_input_pass) (cinfo);

        /* Initialize progress monitoring. */
        if (cinfo->progress != NULL) {
            int nscans;
            /* Estimate number of scans to set pass_limit. */
            if (cinfo->progressive_mode) {
                /* Arbitrarily estimate 2 interleaved DC scans + 3 AC scans/component. */
                nscans = 2 + 3 * cinfo->num_components;
            } else if (cinfo->inputctl->has_multiple_scans) {
                /* For a nonprogressive multiscan file, estimate 1 scan per component. */
                nscans = cinfo->num_components;
            } else {
                nscans = 1;
            }
            cinfo->progress->pass_counter = 0L;
            cinfo->progress->pass_limit = (long) cinfo->total_iMCU_rows * nscans;
            cinfo->progress->completed_passes = 0;
            cinfo->progress->total_passes = 1;
        }
    }
}

```

```

/*
 * jerror.c
 *
 * Copyright (C) 1991-1998, Thomas G. Lane.
 * This file is part of the Independent JPEG Group's software.
 * For conditions of distribution and use, see the accompanying README file.
 *
 * This file contains simple error-reporting and trace-message routines.
 * These are suitable for Unix-like systems and others where writing to
 * stderr is the right thing to do. Many applications will want to replace
 * some or all of these routines.
 *
 * If you define USE_WINDOWS_MESSAGEBOX in jconfig.h or in the makefile,
 * you get a Windows-specific hack to display error messages in a dialog box.
 * It ain't much, but it beats dropping error messages into the bit bucket,
 * which is what happens to output to stderr under most Windows C compilers.
 *
 * These routines are used by both the compression and decompression code.
 */

/* this is not a core library module, so it doesn't define JPEG_INTERNALS */
#include "jinclude.h"
#include "jpeglib.h"
#include "jversion.h"
#include "jerror.h"

#ifdef USE_WINDOWS_MESSAGEBOX
#include <windows.h>
#endif

#ifdef EXIT_FAILURE
/* define exit() codes if not provided */
#define EXIT_FAILURE 1
#endif

/* Create the message string table.
 * We do this from the master message list in jerror.h by re-reading
 * jerror.h with a suitable definition for macro JMESAGE.
 * The message table is made an external symbol just in case any applications
 * want to refer to it directly.
 */

#ifdef NEED_SHORT_EXTERNAL_NAMES
#define jpeg_std_message_table jMsgTable
#endif

#define JMESAGE(code,string) string,

const char * const jpeg_std_message_table[] = {
#include "jerror.h"
  NULL
};

/*
 * Error exit handler: must not return to caller.
 *
 * Applications may override this if they want to get control back after
 * an error. Typically one would longjmp somewhere instead of exiting.
 * The setjmp buffer can be made a private field within an expanded error
 * handler object. Note that the info needed to generate an error message
 * is stored in the error object, so you can generate the message now or
 * later, at your convenience.
 * You should make sure that the JPEG object is cleaned up (with jpeg_abort
 * or jpeg_destroy) at some point.
 */

METHODDEF(void)
error_exit (j_common_ptr cinfo)
{
  /* Always display the message */
  (*cinfo->err->output_message) (cinfo);

  /* Let the memory manager delete any temp files before we die */
  jpeg_destroy(cinfo);

  exit(EXIT_FAILURE);
}

```

```

/*
 * Actual output of an error or trace message.
 * Applications may override this method to send JPEG messages somewhere
 * other than stderr.
 *
 * On Windows, printing to stderr is generally completely useless,
 * so we provide optional code to produce an error-dialog popup.
 * Most Windows applications will still prefer to override this routine,
 * but if they don't, it'll do something at least marginally useful.
 *
 * NOTE: to use the library in an environment that doesn't support the
 * C stdio library, you may have to delete the call to fprintf() entirely,
 * not just not use this routine.
 */

METHODDEF(void)
output_message (j_common_ptr cinfo)
{
    char buffer[JMSG_LENGTH_MAX];

    /* Create the message */
    (*cinfo->err->format_message) (cinfo, buffer);

#ifdef USE_WINDOWS_MESSAGEBOX
    /* Display it in a message dialog box */
    MessageBox(GetActiveWindow(), buffer, "JPEG Library Error",
        MB_OK | MB_ICONERROR);
#else
    /* Send it to stderr, adding a newline */
    fprintf(stderr, "%s\n", buffer);
#endif
}

/*
 * Decide whether to emit a trace or warning message.
 * msg_level is one of:
 *   -1: recoverable corrupt-data warning, may want to abort.
 *   0: important advisory messages (always display to user).
 *   1: first level of tracing detail.
 *   2,3,...: successively more detailed tracing messages.
 * An application might override this method if it wanted to abort on warnings
 * or change the policy about which messages to display.
 */

METHODDEF(void)
emit_message (j_common_ptr cinfo, int msg_level)
{
    struct jpeg_error_mgr * err = cinfo->err;

    if (msg_level < 0) {
        /* It's a warning message. Since corrupt files may generate many warnings,
         * the policy implemented here is to show only the first warning,
         * unless trace_level >= 3.
         */
        if (err->num_warnings == 0 || err->trace_level >= 3)
            (*err->output_message) (cinfo);
        /* Always count warnings in num_warnings. */
        err->num_warnings++;
    } else {
        /* It's a trace message. Show it if trace_level >= msg_level. */
        if (err->trace_level >= msg_level)
            (*err->output_message) (cinfo);
    }
}

/*
 * Format a message string for the most recent JPEG error or message.
 * The message is stored into buffer, which should be at least JMSG_LENGTH_MAX
 * characters. Note that no '\n' character is added to the string.
 * Few applications should need to override this method.
 */

METHODDEF(void)
format_message (j_common_ptr cinfo, char * buffer)
{
    struct jpeg_error_mgr * err = cinfo->err;
    int msg_code = err->msg_code;

```

```

const char * msgtext = NULL;
const char * msgptr;
char ch;
boolean isstring;

/* Look up message string in proper table */
if (msg_code > 0 && msg_code <= err->last_jpeg_message) {
    msgtext = err->jpeg_message_table[msg_code];
} else if (err->addon_message_table != NULL &&
    msg_code >= err->first_addon_message &&
    msg_code <= err->last_addon_message) {
    msgtext = err->addon_message_table[msg_code - err->first_addon_message];
}

/* Defend against bogus message number */
if (msgtext == NULL) {
    err->msg_parm.i[0] = msg_code;
    msgtext = err->jpeg_message_table[0];
}

/* Check for string parameter, as indicated by %s in the message text */
isstring = FALSE;
msgptr = msgtext;
while ((ch = *msgptr++) != '\0') {
    if (ch == '%') {
        if (*msgptr == 's') isstring = TRUE;
        break;
    }
}

/* Format the message into the passed buffer */
if (isstring)
    sprintf(buffer, msgtext, err->msg_parm.s);
else
    sprintf(buffer, msgtext,
        err->msg_parm.i[0], err->msg_parm.i[1],
        err->msg_parm.i[2], err->msg_parm.i[3],
        err->msg_parm.i[4], err->msg_parm.i[5],
        err->msg_parm.i[6], err->msg_parm.i[7]);

/*
 * Reset error state variables at start of a new image.
 * This is called during compression startup to reset trace/error
 * processing to default state, without losing any application-specific
 * method pointers. An application might possibly want to override
 * this method if it has additional error processing state.
 */
METHODDEF(void)
reset_error_mgr (j_common_ptr cinfo)
{
    cinfo->err->num_warnings = 0;
    /* trace_level is not reset since it is an application-supplied parameter */
    cinfo->err->msg_code = 0; /* may be useful as a flag for "no error" */
}

/*
 * Fill in the standard error-handling methods in a jpeg_error_mgr object.
 * Typical call is:
 * struct jpeg_compress_struct cinfo;
 * struct jpeg_error_mgr err;
 *
 * cinfo.err = jpeg_std_error(&err);
 * after which the application may override some of the methods.
 */

GLOBAL(struct jpeg_error_mgr *)
jpeg_std_error (struct jpeg_error_mgr * err)
{
    err->error_exit = error_exit;
    err->emit_message = emit_message;
    err->output_message = output_message;
    err->format_message = format_message;
    err->reset_error_mgr = reset_error_mgr;

    err->trace_level = 0; /* default = no tracing */
    err->num_warnings = 0; /* no warnings emitted yet */
}

```

```
err->msg_code = 0; /* may be useful as a flag for "no error" */
/* Initialize message table pointers */
err->jpeg_message_table = jpeg_std_message_table;
err->last_jpeg_message = (int) JMSG_LASTMSGCODE - 1;

err->addon_message_table = NULL;
err->first_addon_message = 0; /* for safety */
err->last_addon_message = 0;

return err;
```

```

/*
 * jfdctflt.c
 *
 * Copyright (C) 1994-1996, Thomas G. Lane.
 * This file is part of the Independent JPEG Group's software.
 * For conditions of distribution and use, see the accompanying README file.
 *
 * This file contains a floating-point implementation of the
 * forward DCT (Discrete Cosine Transform).
 *
 * This implementation should be more accurate than either of the integer
 * DCT implementations. However, it may not give the same results on all
 * machines because of differences in roundoff behavior. Speed will depend
 * on the hardware's floating point capacity.
 *
 * A 2-D DCT can be done by 1-D DCT on each row followed by 1-D DCT
 * on each column. Direct algorithms are also available, but they are
 * much more complex and seem not to be any faster when reduced to code.
 *
 * This implementation is based on Arai, Agui, and Nakajima's algorithm for
 * scaled DCT. Their original paper (Trans. IEICE E-71(11):1095) is in
 * Japanese, but the algorithm is described in the Pennebaker & Mitchell
 * JPEG textbook (see REFERENCES section in file README). The following code
 * is based directly on figure 4-8 in P&M.
 *
 * While an 8-point DCT cannot be done in less than 11 multiplies, it is
 * possible to arrange the computation so that many of the multiplies are
 * simple scalings of the final outputs. These multiplies can then be
 * folded into the multiplications or divisions by the JPEG quantization
 * table entries. The AA&N method leaves only 5 multiplies and 29 adds
 * to be done in the DCT itself.
 *
 * The primary disadvantage of this method is that with a fixed-point
 * implementation, accuracy is lost due to imprecise representation of the
 * scaled quantization values. However, that problem does not arise if
 * we use floating point arithmetic.
 */

#define JPEG_INTERNALS
#include "jinclude.h"
#include "jpeglib.h"
#include "jdct.h" /* Private declarations for DCT subsystem */

#ifdef DCT_FLOAT_SUPPORTED

/*
 * This module is specialized to the case DCTSIZE = 8.
 */

#if DCTSIZE != 8
  Sorry, this code only copes with 8x8 DCTs. /* deliberate syntax err */
#endif

/*
 * Perform the forward DCT on one block of samples.
 */

GLOBAL(void)
jpeg_fdct_float (FAST_FLOAT * data)
{
  FAST_FLOAT tmp0, tmp1, tmp2, tmp3, tmp4, tmp5, tmp6, tmp7;
  FAST_FLOAT tmp10, tmp11, tmp12, tmp13;
  FAST_FLOAT z1, z2, z3, z4, z5, z11, z13;
  FAST_FLOAT *dataptr;
  int ctr;

  /* Pass 1: process rows. */

  dataptr = data;
  for (ctr = DCTSIZE-1; ctr >= 0; ctr--) {
    tmp0 = dataptr[0] + dataptr[7];
    tmp7 = dataptr[0] - dataptr[7];
    tmp1 = dataptr[1] + dataptr[6];
    tmp6 = dataptr[1] - dataptr[6];
    tmp2 = dataptr[2] + dataptr[5];
    tmp5 = dataptr[2] - dataptr[5];
    tmp3 = dataptr[3] + dataptr[4];
    tmp4 = dataptr[3] - dataptr[4];

    /* Even part */

```



```

tmp10 = tmp0 + tmp3; /* phase 2 */
tmp13 = tmp0 - tmp3;
tmp11 = tmp1 + tmp2;
tmp12 = tmp1 - tmp2;

dataptr[0] = tmp10 + tmp11; /* phase 3 */
dataptr[4] = tmp10 - tmp11;

z1 = (tmp12 + tmp13) * ((FAST_FLOAT) 0.707106781); /* c4 */
dataptr[2] = tmp13 + z1; /* phase 5 */
dataptr[6] = tmp13 - z1;

/* Odd part */

tmp10 = tmp4 + tmp5; /* phase 2 */
tmp11 = tmp5 + tmp6;
tmp12 = tmp6 + tmp7;

/* The rotator is modified from fig 4-8 to avoid extra negations. */
z5 = (tmp10 - tmp12) * ((FAST_FLOAT) 0.382683433); /* c6 */
z2 = ((FAST_FLOAT) 0.541196100) * tmp10 + z5; /* c2-c6 */
z4 = ((FAST_FLOAT) 1.306562965) * tmp12 + z5; /* c2+c6 */
z3 = tmp11 * ((FAST_FLOAT) 0.707106781); /* c4 */

z11 = tmp7 + z3; /* phase 5 */
z13 = tmp7 - z3;

dataptr[5] = z13 + z2; /* phase 6 */
dataptr[3] = z13 - z2;
dataptr[1] = z11 + z4;
dataptr[7] = z11 - z4;

dataptr += DCTSIZE; /* advance pointer to next row */

/* Pass 2: process columns. */
dataptr = data;
for (ctr = DCTSIZE-1; ctr >= 0; ctr--) {
    tmp0 = dataptr[DCTSIZE*0] + dataptr[DCTSIZE*7];
    tmp7 = dataptr[DCTSIZE*0] - dataptr[DCTSIZE*7];
    tmp1 = dataptr[DCTSIZE*1] + dataptr[DCTSIZE*6];
    tmp6 = dataptr[DCTSIZE*1] - dataptr[DCTSIZE*6];
    tmp2 = dataptr[DCTSIZE*2] + dataptr[DCTSIZE*5];
    tmp5 = dataptr[DCTSIZE*2] - dataptr[DCTSIZE*5];
    tmp3 = dataptr[DCTSIZE*3] + dataptr[DCTSIZE*4];
    tmp4 = dataptr[DCTSIZE*3] - dataptr[DCTSIZE*4];

    /* Even part */

    tmp10 = tmp0 + tmp3; /* phase 2 */
    tmp13 = tmp0 - tmp3;
    tmp11 = tmp1 + tmp2;
    tmp12 = tmp1 - tmp2;

    dataptr[DCTSIZE*0] = tmp10 + tmp11; /* phase 3 */
    dataptr[DCTSIZE*4] = tmp10 - tmp11;

    z1 = (tmp12 + tmp13) * ((FAST_FLOAT) 0.707106781); /* c4 */
    dataptr[DCTSIZE*2] = tmp13 + z1; /* phase 5 */
    dataptr[DCTSIZE*6] = tmp13 - z1;

    /* Odd part */

    tmp10 = tmp4 + tmp5; /* phase 2 */
    tmp11 = tmp5 + tmp6;
    tmp12 = tmp6 + tmp7;

    /* The rotator is modified from fig 4-8 to avoid extra negations. */
    z5 = (tmp10 - tmp12) * ((FAST_FLOAT) 0.382683433); /* c6 */
    z2 = ((FAST_FLOAT) 0.541196100) * tmp10 + z5; /* c2-c6 */
    z4 = ((FAST_FLOAT) 1.306562965) * tmp12 + z5; /* c2+c6 */
    z3 = tmp11 * ((FAST_FLOAT) 0.707106781); /* c4 */

    z11 = tmp7 + z3; /* phase 5 */
    z13 = tmp7 - z3;

    dataptr[DCTSIZE*5] = z13 + z2; /* phase 6 */
    dataptr[DCTSIZE*3] = z13 - z2;

```



```

/*
 * jfdctfst.c
 *
 * Copyright (C) 1994-1996, Thomas G. Lane.
 * This file is part of the Independent JPEG Group's software.
 * For conditions of distribution and use, see the accompanying README file.
 *
 * This file contains a fast, not so accurate integer implementation of the
 * forward DCT (Discrete Cosine Transform).
 *
 * A 2-D DCT can be done by 1-D DCT on each row followed by 1-D DCT
 * on each column. Direct algorithms are also available, but they are
 * much more complex and seem not to be any faster when reduced to code.
 *
 * This implementation is based on Arai, Agui, and Nakajima's algorithm for
 * scaled DCT. Their original paper (Trans. IEICE E-71(11):1095) is in
 * Japanese, but the algorithm is described in the Pennebaker & Mitchell
 * JPEG textbook (see REFERENCES section in file README). The following code
 * is based directly on figure 4-8 in P&M.
 * While an 8-point DCT cannot be done in less than 11 multiplies, it is
 * possible to arrange the computation so that many of the multiplies are
 * simple scalings of the final outputs. These multiplies can then be
 * folded into the multiplications or divisions by the JPEG quantization
 * table entries. The AA&N method leaves only 5 multiplies and 29 adds
 * to be done in the DCT itself.
 * The primary disadvantage of this method is that with fixed-point math,
 * accuracy is lost due to imprecise representation of the scaled
 * quantization values. The smaller the quantization table entry, the less
 * precise the scaled value, so this implementation does worse with high-
 * quality-setting files than with low-quality ones.
 */
#define JPEG_INTERNALS
#include "jinclude.h"
#include "jpeglib.h"
#include "jdst.h" /* Private declarations for DCT subsystem */

#ifdef DCT_IFAST_SUPPORTED

/* This module is specialized to the case DCTSIZE = 8.
 */

#if DCTSIZE != 8
Sorry, this code only copes with 8x8 DCTs. /* deliberate syntax err */
#endif

Scaling decisions are generally the same as in the LL&M algorithm;
see jfdctint.c for more details. However, we choose to descale
(right shift) multiplication products as soon as they are formed,
rather than carrying additional fractional bits into subsequent additions.
This compromises accuracy slightly, but it lets us save a few shifts.
More importantly, 16-bit arithmetic is then adequate (for 8-bit samples)
everywhere except in the multiplications proper; this saves a good deal
of work on 16-bit-int machines.

Again to save a few shifts, the intermediate results between pass 1 and
pass 2 are not upsampled, but are represented only to integral precision.

A final compromise is to represent the multiplicative constants to only
8 fractional bits, rather than 13. This saves some shifting work on some
machines, and may also reduce the cost of multiplication (since there
are fewer one-bits in the constants).
*/

#define CONST_BITS 8

/* Some C compilers fail to reduce "FIX(constant)" at compile time, thus
 * causing a lot of useless floating-point operations at run time.
 * To get around this we use the following pre-calculated constants.
 * If you change CONST_BITS you may want to add appropriate values.
 * (With a reasonable C compiler, you can just rely on the FIX() macro...)
 */

#if CONST_BITS == 8
#define FIX_0_382683433 ((INT32) 98) /* FIX(0.382683433) */
#define FIX_0_541196100 ((INT32) 139) /* FIX(0.541196100) */

```

```

#define FIX_0_707106781 ((FIX_0_707106781) 181) /* FIX(0.707106781) */
#define FIX_1_306562965 ((FIX_1_306562965) 334) /* FIX(1.306562965) */
#else
#define FIX_0_382683433 FIX(0.382683433)
#define FIX_0_541196100 FIX(0.541196100)
#define FIX_0_707106781 FIX(0.707106781)
#define FIX_1_306562965 FIX(1.306562965)
#endif

/* We can gain a little more speed, with a further compromise in accuracy,
 * by omitting the addition in a descaling shift. This yields an incorrectly
 * rounded result half the time...
 */

#ifndef USE_ACCURATE_ROUNDING
#undef DESCALE
#define DESCALE(x,n) RIGHT_SHIFT(x, n)
#endif

/* Multiply a DCTELEM variable by an INT32 constant, and immediately
 * descale to yield a DCTELEM result.
 */

#define MULTIPLY(var,const) ((DCTELEM) DESCALE((var) * (const), CONST_BITS))

/*
 * Perform the forward DCT on one block of samples.
 */
GLOBAL(void)
jpeg_fdct_ifast (DCTELEM * data)
{
    DCTELEM tmp0, tmp1, tmp2, tmp3, tmp4, tmp5, tmp6, tmp7;
    DCTELEM tmp10, tmp11, tmp12, tmp13;
    DCTELEM z1, z2, z3, z4, z5, z11, z13;
    DCTELEM *dataptr;
    int ctr;
    SHIFT_TEMPS

    /* Pass 1: process rows. */

    dataptr = data;
    for (ctr = DCTSIZE-1; ctr >= 0; ctr--) {
        tmp0 = dataptr[0] + dataptr[7];
        tmp7 = dataptr[0] - dataptr[7];
        tmp1 = dataptr[1] + dataptr[6];
        tmp6 = dataptr[1] - dataptr[6];
        tmp2 = dataptr[2] + dataptr[5];
        tmp5 = dataptr[2] - dataptr[5];
        tmp3 = dataptr[3] + dataptr[4];
        tmp4 = dataptr[3] - dataptr[4];

        /* Even part */

        tmp10 = tmp0 + tmp3; /* phase 2 */
        tmp13 = tmp0 - tmp3;
        tmp11 = tmp1 + tmp2;
        tmp12 = tmp1 - tmp2;

        dataptr[0] = tmp10 + tmp11; /* phase 3 */
        dataptr[4] = tmp10 - tmp11;

        z1 = MULTIPLY(tmp12 + tmp13, FIX_0_707106781); /* c4 */
        dataptr[2] = tmp13 + z1; /* phase 5 */
        dataptr[6] = tmp13 - z1;

        /* Odd part */

        tmp10 = tmp4 + tmp5; /* phase 2 */
        tmp11 = tmp5 + tmp6;
        tmp12 = tmp6 + tmp7;

        /* The rotator is modified from fig 4-8 to avoid extra negations. */
        z5 = MULTIPLY(tmp10 - tmp12, FIX_0_382683433); /* c6 */
        z2 = MULTIPLY(tmp10, FIX_0_541196100) + z5; /* c2-c6 */
        z4 = MULTIPLY(tmp12, FIX_1_306562965) + z5; /* c2+c6 */
        z3 = MULTIPLY(tmp11, FIX_0_707106781); /* c4 */

```

```

    z11 = tmp7 + z3;          /* phase 5 */
    z13 = tmp7 - z3;

    dataptr[5] = z13 + z2; /* phase 6 */
    dataptr[3] = z13 - z2;
    dataptr[1] = z11 + z4;
    dataptr[7] = z11 - z4;

    dataptr += DCTSIZE; /* advance pointer to next row */
}

/* Pass 2: process columns. */

dataptr = data;
for (ctr = DCTSIZE-1; ctr >= 0; ctr--) {
    tmp0 = dataptr[DCTSIZE*0] + dataptr[DCTSIZE*7];
    tmp7 = dataptr[DCTSIZE*0] - dataptr[DCTSIZE*7];
    tmp1 = dataptr[DCTSIZE*1] + dataptr[DCTSIZE*6];
    tmp6 = dataptr[DCTSIZE*1] - dataptr[DCTSIZE*6];
    tmp2 = dataptr[DCTSIZE*2] + dataptr[DCTSIZE*5];
    tmp5 = dataptr[DCTSIZE*2] - dataptr[DCTSIZE*5];
    tmp3 = dataptr[DCTSIZE*3] + dataptr[DCTSIZE*4];
    tmp4 = dataptr[DCTSIZE*3] - dataptr[DCTSIZE*4];

    /* Even part */

    tmp10 = tmp0 + tmp3; /* phase 2 */
    tmp13 = tmp0 - tmp3;
    tmp11 = tmp1 + tmp2;
    tmp12 = tmp1 - tmp2;

    dataptr[DCTSIZE*0] = tmp10 + tmp11; /* phase 3 */
    dataptr[DCTSIZE*4] = tmp10 - tmp11;

    z1 = MULTIPLY(tmp12 + tmp13, FIX_0_707106781); /* c4 */
    dataptr[DCTSIZE*2] = tmp13 + z1; /* phase 5 */
    dataptr[DCTSIZE*6] = tmp13 - z1;

    /* Odd part */

    tmp10 = tmp4 + tmp5; /* phase 2 */
    tmp11 = tmp5 + tmp6;
    tmp12 = tmp6 + tmp7;

    /* The rotator is modified from fig 4-8 to avoid extra negations. */
    z5 = MULTIPLY(tmp10 - tmp12, FIX_0_382683433); /* c6 */
    z2 = MULTIPLY(tmp10, FIX_0_541196100) + z5; /* c2-c6 */
    z4 = MULTIPLY(tmp12, FIX_1_306562965) + z5; /* c2+c6 */
    z3 = MULTIPLY(tmp11, FIX_0_707106781); /* c4 */

    z11 = tmp7 + z3; /* phase 5 */
    z13 = tmp7 - z3;

    dataptr[DCTSIZE*5] = z13 + z2; /* phase 6 */
    dataptr[DCTSIZE*3] = z13 - z2;
    dataptr[DCTSIZE*1] = z11 + z4;
    dataptr[DCTSIZE*7] = z11 - z4;

    dataptr++; /* advance pointer to next column */
}
}

#endif /* DCT_IFAST_SUPPORTED */

```

```

/*
 * jfdctint.c
 *
 * Copyright (C) 1991-1996, Thomas G. Lane.
 * This file is part of the Independent JPEG Group's software.
 * For conditions of distribution and use, see the accompanying README file.
 *
 * This file contains a slow-but-accurate integer implementation of the
 * forward DCT (Discrete Cosine Transform).
 *
 * A 2-D DCT can be done by 1-D DCT on each row followed by 1-D DCT
 * on each column. Direct algorithms are also available, but they are
 * much more complex and seem not to be any faster when reduced to code.
 *
 * This implementation is based on an algorithm described in
 * C. Loeffler, A. Ligtenberg and G. Moschytz, "Practical Fast 1-D DCT
 * Algorithms with 11 Multiplications", Proc. Int'l. Conf. on Acoustics,
 * Speech, and Signal Processing 1989 (ICASSP '89), pp. 988-991.
 * The primary algorithm described there uses 11 multiplies and 29 adds.
 * We use their alternate method with 12 multiplies and 32 adds.
 * The advantage of this method is that no data path contains more than one
 * multiplication; this allows a very simple and accurate implementation in
 * scaled fixed-point arithmetic, with a minimal number of shifts.
 */

```

```

#define JPEG_INTERNALS
#include "jinclude.h"
#include "jpeglib.h"
#include "jdcct.h" /* Private declarations for DCT subsystem */

```

```

#ifdef DCT_ISLOW_SUPPORTED

```

```

/* This module is specialized to the case DCTSIZE = 8.
 */

```

```

#if DCTSIZE != 8
Sorry, this code only copes with 8x8 DCTs. /* deliberate syntax err */
#endif

```

```

/* The poop on this scaling stuff is as follows:

```

```

Each 1-D DCT step produces outputs which are a factor of sqrt(N)
larger than the true DCT outputs. The final outputs are therefore
a factor of N larger than desired; since N=8 this can be cured by
a simple right shift at the end of the algorithm. The advantage of
this arrangement is that we save two multiplications per 1-D DCT,
because the y0 and y4 outputs need not be divided by sqrt(N).
In the IJG code, this factor of 8 is removed by the quantization step
(in jcdctmgr.c), NOT in this module.

```

```

We have to do addition and subtraction of the integer inputs, which
is no problem, and multiplication by fractional constants, which is
a problem to do in integer arithmetic. We multiply all the constants
by CONST_SCALE and convert them to integer constants (thus retaining
CONST_BITS bits of precision in the constants). After doing a
multiplication we have to divide the product by CONST_SCALE, with proper
rounding, to produce the correct output. This division can be done
cheaply as a right shift of CONST_BITS bits. We postpone shifting
as long as possible so that partial sums can be added together with
full fractional precision.

```

```

The outputs of the first pass are scaled up by PASS1_BITS bits so that
they are represented to better-than-integral precision. These outputs
require BITS_IN_JSAMPLE + PASS1_BITS + 3 bits; this fits in a 16-bit word
with the recommended scaling. (For 12-bit sample data, the intermediate
array is INT32 anyway.)

```

```

To avoid overflow of the 32-bit intermediate results in pass 2, we must
have BITS_IN_JSAMPLE + CONST_BITS + PASS1_BITS <= 26. Error analysis
shows that the values given below are the most effective.

```

```

#if BITS_IN_JSAMPLE == 8
#define CONST_BITS 13
#define PASS1_BITS 2
#else

```

```

#define CONST_BITS 13
#define PASS1_BITS 1 /* use a little precision to avoid overflow */
#endif

/* Some C compilers fail to reduce "FIX(constant)" at compile time, thus
 * causing a lot of useless floating-point operations at run time.
 * To get around this we use the following pre-calculated constants.
 * If you change CONST_BITS you may want to add appropriate values.
 * (With a reasonable C compiler, you can just rely on the FIX() macro...)
 */

#if CONST_BITS == 13
#define FIX_0_298631336 ((INT32) 2446) /* FIX(0.298631336) */
#define FIX_0_390180644 ((INT32) 3196) /* FIX(0.390180644) */
#define FIX_0_541196100 ((INT32) 4433) /* FIX(0.541196100) */
#define FIX_0_765366865 ((INT32) 6270) /* FIX(0.765366865) */
#define FIX_0_899976223 ((INT32) 7373) /* FIX(0.899976223) */
#define FIX_1_175875602 ((INT32) 9633) /* FIX(1.175875602) */
#define FIX_1_501321110 ((INT32) 12299) /* FIX(1.501321110) */
#define FIX_1_847759065 ((INT32) 15137) /* FIX(1.847759065) */
#define FIX_1_961570560 ((INT32) 16069) /* FIX(1.961570560) */
#define FIX_2_053119869 ((INT32) 16819) /* FIX(2.053119869) */
#define FIX_2_562915447 ((INT32) 20995) /* FIX(2.562915447) */
#define FIX_3_072711026 ((INT32) 25172) /* FIX(3.072711026) */
#else
#define FIX_0_298631336 FIX(0.298631336)
#define FIX_0_390180644 FIX(0.390180644)
#define FIX_0_541196100 FIX(0.541196100)
#define FIX_0_765366865 FIX(0.765366865)
#define FIX_0_899976223 FIX(0.899976223)
#define FIX_1_175875602 FIX(1.175875602)
#define FIX_1_501321110 FIX(1.501321110)
#define FIX_1_847759065 FIX(1.847759065)
#define FIX_1_961570560 FIX(1.961570560)
#define FIX_2_053119869 FIX(2.053119869)
#define FIX_2_562915447 FIX(2.562915447)
#define FIX_3_072711026 FIX(3.072711026)
#endif

/* Multiply an INT32 variable by an INT32 constant to yield an INT32 result.
 * For 8-bit samples with the recommended scaling, all the variable
 * and constant values involved are no more than 16 bits wide, so a
 * 16x16->32 bit multiply can be used instead of a full 32x32 multiply.
 * For 12-bit samples, a full 32-bit multiplication will be needed.
 */

#if BITS_IN_JSAMPLE == 8
#define MULTIPLY(var,const) MULTIPLY16C16(var,const)
#else
#define MULTIPLY(var,const) ((var) * (const))
#endif

/* Perform the forward DCT on one block of samples.
 */

GLOBAL(void)
jpeg_fdct_islow (DCTELEM * data)
{
    INT32 tmp0, tmp1, tmp2, tmp3, tmp4, tmp5, tmp6, tmp7;
    INT32 tmp10, tmp11, tmp12, tmp13;
    INT32 z1, z2, z3, z4, z5;
    DCTELEM *dataptr;
    int ctr;
    SHIFT_TEMPS

    /* Pass 1: process rows. */
    /* Note results are scaled up by sqrt(8) compared to a true DCT; */
    /* furthermore, we scale the results by 2**PASS1_BITS. */

    dataptr = data;
    for (ctr = DCTSIZE-1; ctr >= 0; ctr--) {
        tmp0 = dataptr[0] + dataptr[7];
        tmp7 = dataptr[0] - dataptr[7];
        tmp1 = dataptr[1] + dataptr[6];
        tmp6 = dataptr[1] - dataptr[6];
        tmp2 = dataptr[2] + dataptr[5];
        tmp5 = dataptr[2] - dataptr[5];

```

```

tmp3 = dataptr[3] + dataptr[4];
tmp4 = dataptr[3] - dataptr[4];

/* Even part per LL&M figure 1 --- note that published figure is faulty;
 * rotator "sqrt(2)*c1" should be "sqrt(2)*c6".
 */

tmp10 = tmp0 + tmp3;
tmp13 = tmp0 - tmp3;
tmp11 = tmp1 + tmp2;
tmp12 = tmp1 - tmp2;

dataptr[0] = (DCTELEM) ((tmp10 + tmp11) << PASS1_BITS);
dataptr[4] = (DCTELEM) ((tmp10 - tmp11) << PASS1_BITS);

z1 = MULTIPLY(tmp12 + tmp13, FIX_0_541196100);
dataptr[2] = (DCTELEM) DESCALE(z1 + MULTIPLY(tmp13, FIX_0_765366865),
CONST_BITS-PASS1_BITS);
dataptr[6] = (DCTELEM) DESCALE(z1 + MULTIPLY(tmp12, - FIX_1_847759065),
CONST_BITS-PASS1_BITS);

/* Odd part per figure 8 --- note paper omits factor of sqrt(2).
 * cK represents cos(K*pi/16).
 * i0..i3 in the paper are tmp4..tmp7 here.
 */

z1 = tmp4 + tmp7;
z2 = tmp5 + tmp6;
z3 = tmp4 + tmp6;
z4 = tmp5 + tmp7;
z5 = MULTIPLY(z3 + z4, FIX_1_175875602); /* sqrt(2) * c3 */

tmp4 = MULTIPLY(tmp4, FIX_0_298631336); /* sqrt(2) * (-c1+c3+c5-c7) */
tmp5 = MULTIPLY(tmp5, FIX_2_053119869); /* sqrt(2) * ( c1+c3-c5+c7) */
tmp6 = MULTIPLY(tmp6, FIX_3_072711026); /* sqrt(2) * ( c1+c3+c5-c7) */
tmp7 = MULTIPLY(tmp7, FIX_1_501321110); /* sqrt(2) * ( c1+c3-c5-c7) */
z1 = MULTIPLY(z1, - FIX_0_899976223); /* sqrt(2) * (c7-c3) */
z2 = MULTIPLY(z2, - FIX_2_562915447); /* sqrt(2) * (-c1-c3) */
z3 = MULTIPLY(z3, - FIX_1_961570560); /* sqrt(2) * (-c3-c5) */
z4 = MULTIPLY(z4, - FIX_0_390180644); /* sqrt(2) * (c5-c3) */

z3 += z5;
z4 += z5;

dataptr[7] = (DCTELEM) DESCALE(tmp4 + z1 + z3, CONST_BITS-PASS1_BITS);
dataptr[5] = (DCTELEM) DESCALE(tmp5 + z2 + z4, CONST_BITS-PASS1_BITS);
dataptr[3] = (DCTELEM) DESCALE(tmp6 + z2 + z3, CONST_BITS-PASS1_BITS);
dataptr[1] = (DCTELEM) DESCALE(tmp7 + z1 + z4, CONST_BITS-PASS1_BITS);

dataptr += DCTSIZE; /* advance pointer to next row */

/* Pass 2: process columns.
 * We remove the PASS1_BITS scaling, but leave the results scaled up
 * by an overall factor of 8.
 */

dataptr = data;
for (ctr = DCTSIZE-1; ctr >= 0; ctr--) {
    tmp0 = dataptr[DCTSIZE*0] + dataptr[DCTSIZE*7];
    tmp7 = dataptr[DCTSIZE*0] - dataptr[DCTSIZE*7];
    tmp1 = dataptr[DCTSIZE*1] + dataptr[DCTSIZE*6];
    tmp6 = dataptr[DCTSIZE*1] - dataptr[DCTSIZE*6];
    tmp2 = dataptr[DCTSIZE*2] + dataptr[DCTSIZE*5];
    tmp5 = dataptr[DCTSIZE*2] - dataptr[DCTSIZE*5];
    tmp3 = dataptr[DCTSIZE*3] + dataptr[DCTSIZE*4];
    tmp4 = dataptr[DCTSIZE*3] - dataptr[DCTSIZE*4];

    /* Even part per LL&M figure 1 --- note that published figure is faulty;
     * rotator "sqrt(2)*c1" should be "sqrt(2)*c6".
     */

    tmp10 = tmp0 + tmp3;
    tmp13 = tmp0 - tmp3;
    tmp11 = tmp1 + tmp2;
    tmp12 = tmp1 - tmp2;

    dataptr[DCTSIZE*0] = (DCTELEM) DESCALE(tmp10 + tmp11, PASS1_BITS);
    dataptr[DCTSIZE*4] = (DCTELEM) DESCALE(tmp10 - tmp11, PASS1_BITS);

```



```

z1 = MULTIPLY(tmp12 + tmp13, FIX_0_541196100);
dataptr[DCTSIZE*2] = (DCTELEM) DESCALE(z1 + MULTIPLY(tmp13, FIX_1_765366865),
CONST_BITS+PASS1_BITS);
dataptr[DCTSIZE*6] = (DCTELEM) DESCALE(z1 + MULTIPLY(tmp12, - FIX_1_847759065),
CONST_BITS+PASS1_BITS);

```

```

/* Odd part per figure 8 --- note paper omits factor of sqrt(2).
 * cK represents cos(K*pi/16).
 * i0..i3 in the paper are tmp4..tmp7 here.
 */

```

```

z1 = tmp4 + tmp7;
z2 = tmp5 + tmp6;
z3 = tmp4 + tmp6;
z4 = tmp5 + tmp7;
z5 = MULTIPLY(z3 + z4, FIX_1_175875602); /* sqrt(2) * c3 */

```

```

tmp4 = MULTIPLY(tmp4, FIX_0_298631336); /* sqrt(2) * (-c1+c3+c5-c7) */
tmp5 = MULTIPLY(tmp5, FIX_2_053119869); /* sqrt(2) * ( c1+c3-c5+c7) */
tmp6 = MULTIPLY(tmp6, FIX_3_072711026); /* sqrt(2) * ( c1+c3+c5-c7) */
tmp7 = MULTIPLY(tmp7, FIX_1_501321110); /* sqrt(2) * ( c1+c3-c5-c7) */
z1 = MULTIPLY(z1, - FIX_0_899976223); /* sqrt(2) * (c7-c3) */
z2 = MULTIPLY(z2, - FIX_2_562915447); /* sqrt(2) * (-c1-c3) */
z3 = MULTIPLY(z3, - FIX_1_961570560); /* sqrt(2) * (-c3-c5) */
z4 = MULTIPLY(z4, - FIX_0_390180644); /* sqrt(2) * (c5-c3) */

```

```

z3 += z5;
z4 += z5;

```

```

dataptr[DCTSIZE*7] = (DCTELEM) DESCALE(tmp4 + z1 + z3,
CONST_BITS+PASS1_BITS);
dataptr[DCTSIZE*5] = (DCTELEM) DESCALE(tmp5 + z2 + z4,
CONST_BITS+PASS1_BITS);
dataptr[DCTSIZE*3] = (DCTELEM) DESCALE(tmp6 + z2 + z3,
CONST_BITS+PASS1_BITS);
dataptr[DCTSIZE*1] = (DCTELEM) DESCALE(tmp7 + z1 + z4,
CONST_BITS+PASS1_BITS);

```

```

dataptr++; /* advance pointer to next column */

```

```

#endif /* DCT_ISLOW_SUPPORTED */

```

```

/*
 * jidctflt.c
 *
 * Copyright (C) 1994-1998, Thomas G. Lane.
 * This file is part of the Independent JPEG Group's software.
 * For conditions of distribution and use, see the accompanying README file.
 *
 * This file contains a floating-point implementation of the
 * inverse DCT (Discrete Cosine Transform).  In the IJG code, this routine
 * must also perform dequantization of the input coefficients.
 *
 * This implementation should be more accurate than either of the integer
 * IDCT implementations.  However, it may not give the same results on all
 * machines because of differences in roundoff behavior.  Speed will depend
 * on the hardware's floating point capacity.
 *
 * A 2-D IDCT can be done by 1-D IDCT on each column followed by 1-D IDCT
 * on each row (or vice versa, but it's more convenient to emit a row at
 * a time).  Direct algorithms are also available, but they are much more
 * complex and seem not to be any faster when reduced to code.
 *
 * This implementation is based on Arai, Agui, and Nakajima's algorithm for
 * scaled DCT.  Their original paper (Trans. IEICE E-71(11):1095) is in
 * Japanese, but the algorithm is described in the Pennebaker & Mitchell
 * JPEG textbook (see REFERENCES section in file README).  The following code
 * is based directly on figure 4-8 in P&M.
 * While an 8-point DCT cannot be done in less than 11 multiplies, it is
 * possible to arrange the computation so that many of the multiplies are
 * simple scalings of the final outputs.  These multiplies can then be
 * folded into the multiplications or divisions by the JPEG quantization
 * table entries.  The AA&N method leaves only 5 multiplies and 29 adds
 * to be done in the DCT itself.
 * The primary disadvantage of this method is that with a fixed-point
 * implementation, accuracy is lost due to imprecise representation of the
 * scaled quantization values.  However, that problem does not arise if
 * we use floating point arithmetic.
 */

#define JPEG_INTERNALS
#include "jinclude.h"
#include "jpeglib.h"
#include "jdct.h"      /* Private declarations for DCT subsystem */

#ifdef DCT_FLOAT_SUPPORTED

/* This module is specialized to the case DCTSIZE = 8. */

#if DCTSIZE != 8
/* Sorry, this code only copes with 8x8 DCTs.  /* deliberate syntax err */
#endif

/* Dequantize a coefficient by multiplying it by the multiplier-table
 * entry; produce a float result.
 */

#define DEQUANTIZE(coef,quantval)  (((FAST_FLOAT) (coef)) * (quantval))

/*
 * Perform dequantization and inverse DCT on one block of coefficients.
 */

GLOBAL(void)
jpeg_idct_float (j_decompress_ptr cinfo, jpeg_component_info * comp_ptr,
                 JCOEFPTR coef_block,
                 JSAMPARRAY output_buf, JDIMENSION output_col)
{
  FAST_FLOAT tmp0, tmp1, tmp2, tmp3, tmp4, tmp5, tmp6, tmp7;
  FAST_FLOAT tmp10, tmp11, tmp12, tmp13;
  FAST_FLOAT z5, z10, z11, z12, z13;
  JCOEFPTR in_ptr;
  FLOAT_MULT_TYPE * quant_ptr;
  FAST_FLOAT * wsptr;
  JSAMPROW out_ptr;
  JSAMPLE *range_limit = IDCT_range_limit(cinfo);
  int ctr;


```

```
FAST_FLOAT workspace[DCTSIZE*8]; /* buffers data between passes */
SHIFT_TEMPS
```

```
/* Pass 1: process columns from input, store into work array. */
```

```
inptr = coef_block;
quantptr = (FLOAT_MULT_TYPE *) compptr->dct_table;
wsptr = workspace;
for (ctr = DCTSIZE; ctr > 0; ctr--) {
    /* Due to quantization, we will usually find that many of the input
     * coefficients are zero, especially the AC terms. We can exploit this
     * by short-circuiting the IDCT calculation for any column in which all
     * the AC terms are zero. In that case each output is equal to the
     * DC coefficient (with scale factor as needed).
     * With typical images and quantization tables, half or more of the
     * column DCT calculations can be simplified this way.
     */
```

```
if (inptr[DCTSIZE*1] == 0 && inptr[DCTSIZE*2] == 0 &&
    inptr[DCTSIZE*3] == 0 && inptr[DCTSIZE*4] == 0 &&
    inptr[DCTSIZE*5] == 0 && inptr[DCTSIZE*6] == 0 &&
    inptr[DCTSIZE*7] == 0) {
    /* AC terms all zero */
    FAST_FLOAT dval = DEQUANTIZE(inptr[DCTSIZE*0], quantptr[DCTSIZE*0]);
```

```
    wsptr[DCTSIZE*0] = dval;
    wsptr[DCTSIZE*1] = dval;
    wsptr[DCTSIZE*2] = dval;
    wsptr[DCTSIZE*3] = dval;
    wsptr[DCTSIZE*4] = dval;
    wsptr[DCTSIZE*5] = dval;
    wsptr[DCTSIZE*6] = dval;
    wsptr[DCTSIZE*7] = dval;
```

```
    inptr++;          /* advance pointers to next column */
    quantptr++;
    wsptr++;
    continue;
}
```

```
/* Even part */
```

```
tmp0 = DEQUANTIZE(inptr[DCTSIZE*0], quantptr[DCTSIZE*0]);
tmp1 = DEQUANTIZE(inptr[DCTSIZE*2], quantptr[DCTSIZE*2]);
tmp2 = DEQUANTIZE(inptr[DCTSIZE*4], quantptr[DCTSIZE*4]);
tmp3 = DEQUANTIZE(inptr[DCTSIZE*6], quantptr[DCTSIZE*6]);
```

```
tmp10 = tmp0 + tmp2;    /* phase 3 */
tmp11 = tmp0 - tmp2;
```

```
tmp13 = tmp1 + tmp3;    /* phases 5-3 */
tmp12 = (tmp1 - tmp3) * ((FAST_FLOAT) 1.414213562) - tmp13; /* 2*c4 */
```

```
tmp0 = tmp10 + tmp13;   /* phase 2 */
tmp3 = tmp10 - tmp13;
tmp1 = tmp11 + tmp12;
tmp2 = tmp11 - tmp12;
```

```
/* Odd part */
```

```
tmp4 = DEQUANTIZE(inptr[DCTSIZE*1], quantptr[DCTSIZE*1]);
tmp5 = DEQUANTIZE(inptr[DCTSIZE*3], quantptr[DCTSIZE*3]);
tmp6 = DEQUANTIZE(inptr[DCTSIZE*5], quantptr[DCTSIZE*5]);
tmp7 = DEQUANTIZE(inptr[DCTSIZE*7], quantptr[DCTSIZE*7]);
```

```
z13 = tmp6 + tmp5;      /* phase 6 */
z10 = tmp6 - tmp5;
z11 = tmp4 + tmp7;
z12 = tmp4 - tmp7;
```

```
tmp7 = z11 + z13;       /* phase 5 */
tmp11 = (z11 - z13) * ((FAST_FLOAT) 1.414213562); /* 2*c4 */
```

```
z5 = (z10 + z12) * ((FAST_FLOAT) 1.847759065); /* 2*c2 */
tmp10 = ((FAST_FLOAT) 1.082392200) * z12 - z5; /* 2*(c2-c6) */
tmp12 = ((FAST_FLOAT) -2.613125930) * z10 + z5; /* -2*(c2+c6) */
```

```
tmp6 = tmp12 - tmp7;    /* phase 2 */
tmp5 = tmp11 - tmp6;
tmp4 = tmp10 + tmp5;
```

```

    wsptr[DCTSIZE*0] = tmp0 - tmp7;
    wsptr[DCTSIZE*7] = tmp0 - tmp7;
    wsptr[DCTSIZE*1] = tmp1 + tmp6;
    wsptr[DCTSIZE*6] = tmp1 - tmp6;
    wsptr[DCTSIZE*2] = tmp2 + tmp5;
    wsptr[DCTSIZE*5] = tmp2 - tmp5;
    wsptr[DCTSIZE*4] = tmp3 + tmp4;
    wsptr[DCTSIZE*3] = tmp3 - tmp4;

    inptr++;          /* advance pointers to next column */
    quantptr++;
    wsptr++;
}

/* Pass 2: process rows from work array, store into output array. */
/* Note that we must descale the results by a factor of 8 == 2**3. */

wsptr = workspace;
for (ctr = 0; ctr < DCTSIZE; ctr++) {
    outptr = output_buf[ctr] + output_col;
    /* Rows of zeroes can be exploited in the same way as we did with columns.
     * However, the column calculation has created many nonzero AC terms, so
     * the simplification applies less often (typically 5% to 10% of the time).
     * And testing floats for zero is relatively expensive, so we don't bother.
     */

    /* Even part */

    tmp10 = wsptr[0] + wsptr[4];
    tmp11 = wsptr[0] - wsptr[4];

    tmp13 = wsptr[2] + wsptr[6];
    tmp12 = (wsptr[2] - wsptr[6]) * ((FAST_FLOAT) 1.414213562) - tmp13;

    tmp0 = tmp10 + tmp13;
    tmp3 = tmp10 - tmp13;
    tmp1 = tmp11 + tmp12;
    tmp2 = tmp11 - tmp12;

    /* Odd part */

    z13 = wsptr[5] + wsptr[3];
    z10 = wsptr[5] - wsptr[3];
    z11 = wsptr[1] + wsptr[7];
    z12 = wsptr[1] - wsptr[7];

    tmp7 = z11 + z13;
    tmp11 = (z11 - z13) * ((FAST_FLOAT) 1.414213562);

    z5 = (z10 + z12) * ((FAST_FLOAT) 1.847759065); /* 2*c2 */
    tmp10 = ((FAST_FLOAT) 1.082392200) * z12 - z5; /* 2*(c2-c6) */
    tmp12 = ((FAST_FLOAT) -2.613125930) * z10 + z5; /* -2*(c2+c6) */

    tmp6 = tmp12 - tmp7;
    tmp5 = tmp11 - tmp6;
    tmp4 = tmp10 + tmp5;

    /* Final output stage: scale down by a factor of 8 and range-limit */

    outptr[0] = range_limit[(int) DESCALE((INT32) (tmp0 + tmp7), 3)
        & RANGE_MASK];
    outptr[7] = range_limit[(int) DESCALE((INT32) (tmp0 - tmp7), 3)
        & RANGE_MASK];
    outptr[1] = range_limit[(int) DESCALE((INT32) (tmp1 + tmp6), 3)
        & RANGE_MASK];
    outptr[6] = range_limit[(int) DESCALE((INT32) (tmp1 - tmp6), 3)
        & RANGE_MASK];
    outptr[2] = range_limit[(int) DESCALE((INT32) (tmp2 + tmp5), 3)
        & RANGE_MASK];
    outptr[5] = range_limit[(int) DESCALE((INT32) (tmp2 - tmp5), 3)
        & RANGE_MASK];
    outptr[4] = range_limit[(int) DESCALE((INT32) (tmp3 + tmp4), 3)
        & RANGE_MASK];
    outptr[3] = range_limit[(int) DESCALE((INT32) (tmp3 - tmp4), 3)
        & RANGE_MASK];

    wsptr += DCTSIZE;    /* advance pointer to next row */
}

```



```

/*
 * jidctfst.c
 *
 * Copyright (C) 1994-1998, Thomas G. Lane.
 * This file is part of the Independent JPEG Group's software.
 * For conditions of distribution and use, see the accompanying README file.
 *
 * This file contains a fast, not so accurate integer implementation of the
 * inverse DCT (Discrete Cosine Transform). In the IJG code, this routine
 * must also perform dequantization of the input coefficients.
 *
 * A 2-D IDCT can be done by 1-D IDCT on each column followed by 1-D IDCT
 * on each row (or vice versa, but it's more convenient to emit a row at
 * a time). Direct algorithms are also available, but they are much more
 * complex and seem not to be any faster when reduced to code.
 *
 * This implementation is based on Arai, Agui, and Nakajima's algorithm for
 * scaled DCT. Their original paper (Trans. IEICE E-71(11):1095) is in
 * Japanese, but the algorithm is described in the Pennebaker & Mitchell
 * JPEG textbook (see REFERENCES section in file README). The following code
 * is based directly on figure 4-8 in P&M.
 *
 * While an 8-point DCT cannot be done in less than 11 multiplies, it is
 * possible to arrange the computation so that many of the multiplies are
 * simple scalings of the final outputs. These multiplies can then be
 * folded into the multiplications or divisions by the JPEG quantization
 * table entries. The AA&N method leaves only 5 multiplies and 29 adds
 * to be done in the DCT itself.
 *
 * The primary disadvantage of this method is that with fixed-point math,
 * accuracy is lost due to imprecise representation of the scaled
 * quantization values. The smaller the quantization table entry, the less
 * precise the scaled value, so this implementation does worse with high-
 * quality-setting files than with low-quality ones.
 */

#define JPEG_INTERNALS
#include "jinclude.h"
#include "jpeglib.h"
#include "jdct.h" /* Private declarations for DCT subsystem */

#ifdef DCT_IFAST_SUPPORTED

/* This module is specialized to the case DCTSIZE = 8.
 */

#if DCTSIZE != 8
/* Sorry, this code only copes with 8x8 DCTs. */ /* deliberate syntax err */
#endif

/* Scaling decisions are generally the same as in the LL&M algorithm;
 * see jidctint.c for more details. However, we choose to descale
 * (right shift) multiplication products as soon as they are formed,
 * rather than carrying additional fractional bits into subsequent additions.
 * This compromises accuracy slightly, but it lets us save a few shifts.
 * More importantly, 16-bit arithmetic is then adequate (for 8-bit samples)
 * everywhere except in the multiplications proper; this saves a good deal
 * of work on 16-bit-int machines.
 *
 * The dequantized coefficients are not integers because the AA&N scaling
 * factors have been incorporated. We represent them scaled up by PASS1_BITS,
 * so that the first and second IDCT rounds have the same input scaling.
 * For 8-bit JSAMPLEs, we choose IFAST_SCALE_BITS = PASS1_BITS so as to
 * avoid a descaling shift; this compromises accuracy rather drastically
 * for small quantization table entries, but it saves a lot of shifts.
 * For 12-bit JSAMPLEs, there's no hope of using 16x16 multiplies anyway,
 * so we use a much larger scaling factor to preserve accuracy.
 *
 * A final compromise is to represent the multiplicative constants to only
 * 8 fractional bits, rather than 13. This saves some shifting work on some
 * machines, and may also reduce the cost of multiplication (since there
 * are fewer one-bits in the constants).
 */

#if BITS_IN_JSAMPLE == 8
#define CONST_BITS 8
#define PASS1_BITS 2
#else
#define CONST_BITS 8
#endif

```

```

#define PASS1_BITS 1 /* lose a little precision to avoid overflow */
#endif

/* Some C compilers fail to reduce "FIX(constant)" at compile time, thus
 * causing a lot of useless floating-point operations at run time.
 * To get around this we use the following pre-calculated constants.
 * If you change CONST_BITS you may want to add appropriate values.
 * (With a reasonable C compiler, you can just rely on the FIX() macro...)
 */

#if CONST_BITS == 8
#define FIX_1_082392200 ((INT32) 277) /* FIX(1.082392200) */
#define FIX_1_414213562 ((INT32) 362) /* FIX(1.414213562) */
#define FIX_1_847759065 ((INT32) 473) /* FIX(1.847759065) */
#define FIX_2_613125930 ((INT32) 669) /* FIX(2.613125930) */
#else
#define FIX_1_082392200 FIX(1.082392200)
#define FIX_1_414213562 FIX(1.414213562)
#define FIX_1_847759065 FIX(1.847759065)
#define FIX_2_613125930 FIX(2.613125930)
#endif

/* We can gain a little more speed, with a further compromise in accuracy,
 * by omitting the addition in a descaling shift. This yields an incorrectly
 * rounded result half the time...
 */

#ifndef USE_ACCURATE_ROUNDING
#undef DESCALE
#define DESCALE(x,n) RIGHT_SHIFT(x, n)
#endif

/* Multiply a DCTELEM variable by an INT32 constant, and immediately
 * descale to yield a DCTELEM result.
 */
#define MULTIPLY(var,const) ((DCTELEM) DESCALE((var) * (const), CONST_BITS))

/* Dequantize a coefficient by multiplying it by the multiplier-table
 * entry; produce a DCTELEM result. For 8-bit data a 16x16->16
 * multiplication will do. For 12-bit data, the multiplier table is
 * declared INT32, so a 32-bit multiply will be used.
 */
#if BITS_IN_JSAMPLE == 8
#define DEQUANTIZE(coef,quantval) (((IFAST_MULT_TYPE) (coef)) * (quantval))
#else
#define DEQUANTIZE(coef,quantval) \
    DESCALE((coef)*(quantval), IFAST_SCALE_BITS-PASS1_BITS)
#endif

/* Like DESCALE, but applies to a DCTELEM and produces an int.
 * We assume that int right shift is unsigned if INT32 right shift is.
 */

#ifndef RIGHT_SHIFT_IS_UNSIGNED
#define ISHIFT_TEMPS DCTELEM ishift_temp;
#if BITS_IN_JSAMPLE == 8
#define DCTELEM_BITS 16 /* DCTELEM may be 16 or 32 bits */
#else
#define DCTELEM_BITS 32 /* DCTELEM must be 32 bits */
#endif
#define IRIGHT_SHIFT(x,shft) \
    ((ishift_temp = (x)) < 0 ? \
     (ishift_temp >> (shft)) | ((~(DCTELEM) 0) << (DCTELEM_BITS-(shft))) : \
     (ishift_temp >> (shft)))
#else
#define ISHIFT_TEMPS
#define IRIGHT_SHIFT(x,shft) ((x) >> (shft))
#endif

#ifdef USE_ACCURATE_ROUNDING
#define IDESCALE(x,n) ((int) IRIGHT_SHIFT((x) + (1 << ((n)-1)), n))
#else
#define IDESCALE(x,n) ((int) IRIGHT_SHIFT(x, n))
#endif

```

```

/*
 * Perform dequantization and inverse DCT on one block of coefficients.
 */

GLOBAL(void)
jpeg_idct_ifast (j_decompress_ptr cinfo, jpeg_component_info * compptr,
                 JCOEFPTR coef_block,
                 JSAMPARRAY output_buf, JDIMENSION output_col)
{
    DCTELEM tmp0, tmp1, tmp2, tmp3, tmp4, tmp5, tmp6, tmp7;
    DCTELEM tmp10, tmp11, tmp12, tmp13;
    DCTELEM z5, z10, z11, z12, z13;
    JCOEFPTR inptr;
    IFAST_MULT_TYPE * quantptr;
    int * wsptr;
    JSAMPROW outptr;
    JSAMPLE *range_limit = IDCT_range_limit(cinfo);
    int ctr;
    int workspace[DCTSIZE2]; /* buffers data between passes */
    SHIFT_TEMPS /* for DESCALE */
    ISHIFT_TEMPS /* for IDESCALE */

    /* Pass 1: process columns from input, store into work array. */

    inptr = coef_block;
    quantptr = (IFAST_MULT_TYPE *) compptr->dct_table;
    wsptr = workspace;
    for (ctr = DCTSIZE; ctr > 0; ctr--) {
        /* Due to quantization, we will usually find that many of the input
         * coefficients are zero, especially the AC terms. We can exploit this
         * by short-circuiting the IDCT calculation for any column in which all
         * the AC terms are zero. In that case each output is equal to the
         * DC coefficient (with scale factor as needed).
         * With typical images and quantization tables, half or more of the
         * column DCT calculations can be simplified this way.
         */

        if (inptr[DCTSIZE*1] == 0 && inptr[DCTSIZE*2] == 0 &&
            inptr[DCTSIZE*3] == 0 && inptr[DCTSIZE*4] == 0 &&
            inptr[DCTSIZE*5] == 0 && inptr[DCTSIZE*6] == 0 &&
            inptr[DCTSIZE*7] == 0) {
            /* AC terms all zero */
            int dval = (int) DEQUANTIZE(inptr[DCTSIZE*0], quantptr[DCTSIZE*0]);

            wsptr[DCTSIZE*0] = dval;
            wsptr[DCTSIZE*1] = dval;
            wsptr[DCTSIZE*2] = dval;
            wsptr[DCTSIZE*3] = dval;
            wsptr[DCTSIZE*4] = dval;
            wsptr[DCTSIZE*5] = dval;
            wsptr[DCTSIZE*6] = dval;
            wsptr[DCTSIZE*7] = dval;

            inptr++; /* advance pointers to next column */
            quantptr++;
            wsptr++;
            continue;
        }

        /* Even part */

        tmp0 = DEQUANTIZE(inptr[DCTSIZE*0], quantptr[DCTSIZE*0]);
        tmp1 = DEQUANTIZE(inptr[DCTSIZE*2], quantptr[DCTSIZE*2]);
        tmp2 = DEQUANTIZE(inptr[DCTSIZE*4], quantptr[DCTSIZE*4]);
        tmp3 = DEQUANTIZE(inptr[DCTSIZE*6], quantptr[DCTSIZE*6]);

        tmp10 = tmp0 + tmp2; /* phase 3 */
        tmp11 = tmp0 - tmp2;

        tmp13 = tmp1 + tmp3; /* phases 5-3 */
        tmp12 = MULTIPLY(tmp1 - tmp3, FIX_1_414213562) - tmp13; /* 2*c4 */

        tmp0 = tmp10 + tmp13; /* phase 2 */
        tmp3 = tmp10 - tmp13;
        tmp1 = tmp11 + tmp12;
        tmp2 = tmp11 - tmp12;

        /* Odd part */
    }
}

```



```

tmp4 = DEQUANTIZE(inpstr[DCTSIZE*1], quantptr[DCTSIZE*1]);
tmp5 = DEQUANTIZE(inpstr[DCTSIZE*3], quantptr[DCTSIZE*3]);
tmp6 = DEQUANTIZE(inpstr[DCTSIZE*5], quantptr[DCTSIZE*5]);
tmp7 = DEQUANTIZE(inpstr[DCTSIZE*7], quantptr[DCTSIZE*7]);

z13 = tmp6 + tmp5;      /* phase 6 */
z10 = tmp6 - tmp5;
z11 = tmp4 + tmp7;
z12 = tmp4 - tmp7;

tmp7 = z11 + z13;      /* phase 5 */
tmp11 = MULTIPLY(z11 - z13, FIX_1_414213562); /* 2*c4 */

z5 = MULTIPLY(z10 + z12, FIX_1_847759065); /* 2*c2 */
tmp10 = MULTIPLY(z12, FIX_1_082392200) - z5; /* 2*(c2-c6) */
tmp12 = MULTIPLY(z10, -FIX_2_613125930) + z5; /* -2*(c2+c6) */

tmp6 = tmp12 - tmp7;    /* phase 2 */
tmp5 = tmp11 - tmp6;
tmp4 = tmp10 + tmp5;

wsptr[DCTSIZE*0] = (int) (tmp0 + tmp7);
wsptr[DCTSIZE*7] = (int) (tmp0 - tmp7);
wsptr[DCTSIZE*1] = (int) (tmp1 + tmp6);
wsptr[DCTSIZE*6] = (int) (tmp1 - tmp6);
wsptr[DCTSIZE*2] = (int) (tmp2 + tmp5);
wsptr[DCTSIZE*5] = (int) (tmp2 - tmp5);
wsptr[DCTSIZE*4] = (int) (tmp3 + tmp4);
wsptr[DCTSIZE*3] = (int) (tmp3 - tmp4);

inpstr++;      /* advance pointers to next column */
quantptr++;
wsptr++;

/* Pass 2: process rows from work array, store into output array. */
/* Note that we must descale the results by a factor of 8 == 2**3, */
/* and also undo the PASS1_BITS scaling. */

wsptr = workspace;
for (ctr = 0; ctr < DCTSIZE; ctr++) {
    outptr = output_buf[ctr] + output_col;
    /* Rows of zeroes can be exploited in the same way as we did with columns.
     * However, the column calculation has created many nonzero AC terms, so
     * the simplification applies less often (typically 5% to 10% of the time).
     * On machines with very fast multiplication, it's possible that the
     * test takes more time than it's worth. In that case this section
     * may be commented out.
     */
    #ifndef NO_ZERO_ROW_TEST
    if (wsptr[1] == 0 && wsptr[2] == 0 && wsptr[3] == 0 && wsptr[4] == 0 &&
        wsptr[5] == 0 && wsptr[6] == 0 && wsptr[7] == 0) {
        /* AC terms all zero */
        JSAMPLE dcvall = range_limit[IDESCALE(wsptr[0], PASS1_BITS+3)
            & RANGE_MASK];

        outptr[0] = dcvall;
        outptr[1] = dcvall;
        outptr[2] = dcvall;
        outptr[3] = dcvall;
        outptr[4] = dcvall;
        outptr[5] = dcvall;
        outptr[6] = dcvall;
        outptr[7] = dcvall;

        wsptr += DCTSIZE;      /* advance pointer to next row */
        continue;
    }
    #endif

    /* Even part */

    tmp10 = ((DCTELEM) wsptr[0] + (DCTELEM) wsptr[4]);
    tmp11 = ((DCTELEM) wsptr[0] - (DCTELEM) wsptr[4]);

    tmp13 = ((DCTELEM) wsptr[2] + (DCTELEM) wsptr[6]);
    tmp12 = MULTIPLY((DCTELEM) wsptr[2] - (DCTELEM) wsptr[6], FIX_1_414213562)
        - tmp13;

```

```

tmp0 = tmp10 + tmp13;
tmp3 = tmp10 - tmp13;
tmp1 = tmp11 + tmp12;
tmp2 = tmp11 - tmp12;

/* Odd part */

z13 = (DCTELEM) wsptr[5] + (DCTELEM) wsptr[3];
z10 = (DCTELEM) wsptr[5] - (DCTELEM) wsptr[3];
z11 = (DCTELEM) wsptr[1] + (DCTELEM) wsptr[7];
z12 = (DCTELEM) wsptr[1] - (DCTELEM) wsptr[7];

tmp7 = z11 + z13;          /* phase 5 */
tmp11 = MULTIPLY(z11 - z13, FIX_1_414213562); /* 2*c4 */

z15 = MULTIPLY(z10 + z12, FIX_1_847759065); /* 2*c2 */
tmp10 = MULTIPLY(z12, FIX_1_082392200) - z5; /* 2*(c2-c6) */
tmp12 = MULTIPLY(z10, - FIX_2_613125930) + z5; /* -2*(c2+c6) */

tmp6 = tmp12 - tmp7;       /* phase 2 */
tmp5 = tmp11 - tmp6;
tmp4 = tmp10 + tmp5;

/* Final output stage: scale down by a factor of 8 and range-limit */
outptr[0] = range_limit[IDESCALE(tmp0 + tmp7, PASS1_BITS+3)
    & RANGE_MASK];
outptr[7] = range_limit[IDESCALE(tmp0 - tmp7, PASS1_BITS+3)
    & RANGE_MASK];
outptr[1] = range_limit[IDESCALE(tmp1 + tmp6, PASS1_BITS+3)
    & RANGE_MASK];
outptr[6] = range_limit[IDESCALE(tmp1 - tmp6, PASS1_BITS+3)
    & RANGE_MASK];
outptr[2] = range_limit[IDESCALE(tmp2 + tmp5, PASS1_BITS+3)
    & RANGE_MASK];
outptr[5] = range_limit[IDESCALE(tmp2 - tmp5, PASS1_BITS+3)
    & RANGE_MASK];
outptr[4] = range_limit[IDESCALE(tmp3 + tmp4, PASS1_BITS+3)
    & RANGE_MASK];
outptr[3] = range_limit[IDESCALE(tmp3 - tmp4, PASS1_BITS+3)
    & RANGE_MASK];

wsptr += DCTSIZE;          /* advance pointer to next row */

#endif /* DCT_IFAST_SUPPORTED */

```

```

/*
 * jidctint.c
 *
 * Copyright (C) 1991-1998, Thomas G. Lane.
 * This file is part of the Independent JPEG Group's software.
 * For conditions of distribution and use, see the accompanying README file.
 *
 * This file contains a slow-but-accurate integer implementation of the
 * inverse DCT (Discrete Cosine Transform).  In the IJG code, this routine
 * must also perform dequantization of the input coefficients.
 *
 * A 2-D IDCT can be done by 1-D IDCT on each column followed by 1-D IDCT
 * on each row (or vice versa, but it's more convenient to emit a row at
 * a time).  Direct algorithms are also available, but they are much more
 * complex and seem not to be any faster when reduced to code.
 *
 * This implementation is based on an algorithm described in
 * C. Loeffler, A. Ligtenberg and G. Moschytz, "Practical Fast 1-D DCT
 * Algorithms with 11 Multiplications", Proc. Int'l. Conf. on Acoustics,
 * Speech, and Signal Processing 1989 (ICASSP '89), pp. 988-991.
 * The primary algorithm described there uses 11 multiplies and 29 adds.
 * We use their alternate method with 12 multiplies and 32 adds.
 * The advantage of this method is that no data path contains more than one
 * multiplication; this allows a very simple and accurate implementation in
 * scaled fixed-point arithmetic, with a minimal number of shifts.
 */

#define JPEG_INTERNALS
#include "jinclude.h"
#include "jpeglib.h"
#include "jdct.h"          /* Private declarations for DCT subsystem */

#ifdef DCT_ISLOW_SUPPORTED

/*
 * This module is specialized to the case DCTSIZE = 8.
 */

#if DCTSIZE != 8
/* Sorry, this code only copes with 8x8 DCTs.  /* deliberate syntax err */
#endif

/*
 * The poop on this scaling stuff is as follows:
 *
 * Each 1-D IDCT step produces outputs which are a factor of sqrt(N)
 * larger than the true IDCT outputs.  The final outputs are therefore
 * a factor of N larger than desired; since N=8 this can be cured by
 * a simple right shift at the end of the algorithm.  The advantage of
 * this arrangement is that we save two multiplications per 1-D IDCT,
 * because the y0 and y4 inputs need not be divided by sqrt(N).
 *
 * We have to do addition and subtraction of the integer inputs, which
 * is no problem, and multiplication by fractional constants, which is
 * a problem to do in integer arithmetic.  We multiply all the constants
 * by CONST_SCALE and convert them to integer constants (thus retaining
 * CONST_BITS bits of precision in the constants).  After doing a
 * multiplication we have to divide the product by CONST_SCALE, with proper
 * rounding, to produce the correct output.  This division can be done
 * cheaply as a right shift of CONST_BITS bits.  We postpone shifting
 * as long as possible so that partial sums can be added together with
 * full fractional precision.
 *
 * The outputs of the first pass are scaled up by PASS1_BITS bits so that
 * they are represented to better-than-integral precision.  These outputs
 * require BITS_IN_JSAMPLE + PASS1_BITS + 3 bits; this fits in a 16-bit word
 * with the recommended scaling.  (To scale up 12-bit sample data further, an
 * intermediate INT32 array would be needed.)
 *
 * To avoid overflow of the 32-bit intermediate results in pass 2, we must
 * have BITS_IN_JSAMPLE + CONST_BITS + PASS1_BITS <= 26.  Error analysis
 * shows that the values given below are the most effective.
 */

#if BITS_IN_JSAMPLE == 8
#define CONST_BITS 13
#define PASS1_BITS 2
#else

```

```

#define CONST_BITS 13
#define PASS1_BITS 1 /* Use a little precision to avoid overflow */
#endif

/* Some C compilers fail to reduce "FIX(constant)" at compile time, thus
 * causing a lot of useless floating-point operations at run time.
 * To get around this we use the following pre-calculated constants.
 * If you change CONST_BITS you may want to add appropriate values.
 * (With a reasonable C compiler, you can just rely on the FIX() macro...)
 */

#if CONST_BITS == 13
#define FIX_0_298631336 ((INT32) 2446) /* FIX(0.298631336) */
#define FIX_0_390180644 ((INT32) 3196) /* FIX(0.390180644) */
#define FIX_0_541196100 ((INT32) 4433) /* FIX(0.541196100) */
#define FIX_0_765366865 ((INT32) 6270) /* FIX(0.765366865) */
#define FIX_0_899976223 ((INT32) 7373) /* FIX(0.899976223) */
#define FIX_1_175875602 ((INT32) 9633) /* FIX(1.175875602) */
#define FIX_1_501321110 ((INT32) 12299) /* FIX(1.501321110) */
#define FIX_1_847759065 ((INT32) 15137) /* FIX(1.847759065) */
#define FIX_1_961570560 ((INT32) 16069) /* FIX(1.961570560) */
#define FIX_2_053119869 ((INT32) 16819) /* FIX(2.053119869) */
#define FIX_2_562915447 ((INT32) 20995) /* FIX(2.562915447) */
#define FIX_3_072711026 ((INT32) 25172) /* FIX(3.072711026) */
#else
#define FIX_0_298631336 FIX(0.298631336)
#define FIX_0_390180644 FIX(0.390180644)
#define FIX_0_541196100 FIX(0.541196100)
#define FIX_0_765366865 FIX(0.765366865)
#define FIX_0_899976223 FIX(0.899976223)
#define FIX_1_175875602 FIX(1.175875602)
#define FIX_1_501321110 FIX(1.501321110)
#define FIX_1_847759065 FIX(1.847759065)
#define FIX_1_961570560 FIX(1.961570560)
#define FIX_2_053119869 FIX(2.053119869)
#define FIX_2_562915447 FIX(2.562915447)
#define FIX_3_072711026 FIX(3.072711026)
#endif

/*
 * Multiply an INT32 variable by an INT32 constant to yield an INT32 result.
 * For 8-bit samples with the recommended scaling, all the variable
 * and constant values involved are no more than 16 bits wide, so a
 * 16x16->32 bit multiply can be used instead of a full 32x32 multiply.
 * For 12-bit samples, a full 32-bit multiplication will be needed.
 */

#if BITS_IN_JSAMPLE == 8
#define MULTIPLY(var, const) MULTIPLY16C16(var, const)
#else
#define MULTIPLY(var, const) ((var) * (const))
#endif

/* Dequantize a coefficient by multiplying it by the multiplier-table
 * entry; produce an int result. In this module, both inputs and result
 * are 16 bits or less, so either int or short multiply will work.
 */

#define DEQUANTIZE(coef, quantval) (((ISLOW_MULT_TYPE) (coef)) * (quantval))

/*
 * Perform dequantization and inverse DCT on one block of coefficients.
 */

GLOBAL(void)
jpeg_idct_islow (j_decompress_ptr cinfo, jpeg_component_info * comp_ptr,
                 JCOEFPTR coef_block,
                 JSAMPARRAY output_buf, JDIMENSION output_col)
{
    INT32 tmp0, tmp1, tmp2, tmp3;
    INT32 tmp10, tmp11, tmp12, tmp13;
    INT32 z1, z2, z3, z4, z5;
    JCOEFPTR in_ptr;
    ISLOW_MULT_TYPE * quant_ptr;
    int * wsptr;
    JSAMPROW out_ptr;
    JSAMPLE *range_limit = IDCT_range_limit(cinfo);
    int ctr;

```

```

int workspace[DCTSIZE2]; /* buffers data between passes */
SHIFT_TEMPS

/* Pass 1: process columns from input, store into work array. */
/* Note results are scaled up by sqrt(8) compared to a true IDCT; */
/* furthermore, we scale the results by 2**PASS1_BITS. */

inptr = coef_block;
quantptr = (ISLOW_MULT_TYPE *) compptr->dct_table;
wsptr = workspace;
for (ctr = DCTSIZE; ctr > 0; ctr--) {
    /* Due to quantization, we will usually find that many of the input
     * coefficients are zero, especially the AC terms. We can exploit this
     * by short-circuiting the IDCT calculation for any column in which all
     * the AC terms are zero. In that case each output is equal to the
     * DC coefficient (with scale factor as needed).
     * With typical images and quantization tables, half or more of the
     * column DCT calculations can be simplified this way.
     */

    if (inptr[DCTSIZE*1] == 0 && inptr[DCTSIZE*2] == 0 &&
        inptr[DCTSIZE*3] == 0 && inptr[DCTSIZE*4] == 0 &&
        inptr[DCTSIZE*5] == 0 && inptr[DCTSIZE*6] == 0 &&
        inptr[DCTSIZE*7] == 0) {
        /* AC terms all zero */
        int dval = DEQUANTIZE(inptr[DCTSIZE*0], quantptr[DCTSIZE*0]) << PASS1_BITS;

        wsptr[DCTSIZE*0] = dval;
        wsptr[DCTSIZE*1] = dval;
        wsptr[DCTSIZE*2] = dval;
        wsptr[DCTSIZE*3] = dval;
        wsptr[DCTSIZE*4] = dval;
        wsptr[DCTSIZE*5] = dval;
        wsptr[DCTSIZE*6] = dval;
        wsptr[DCTSIZE*7] = dval;

        inptr++;          /* advance pointers to next column */
        quantptr++;
        wsptr++;
        continue;
    }

    /* Even part: reverse the even part of the forward DCT. */
    /* The rotator is sqrt(2)*c(-6). */

    z2 = DEQUANTIZE(inptr[DCTSIZE*2], quantptr[DCTSIZE*2]);
    z3 = DEQUANTIZE(inptr[DCTSIZE*6], quantptr[DCTSIZE*6]);

    z1 = MULTIPLY(z2 + z3, FIX_0_541196100);
    tmp2 = z1 + MULTIPLY(z3, -FIX_1_847759065);
    tmp3 = z1 + MULTIPLY(z2, FIX_0_765366865);

    z2 = DEQUANTIZE(inptr[DCTSIZE*0], quantptr[DCTSIZE*0]);
    z3 = DEQUANTIZE(inptr[DCTSIZE*4], quantptr[DCTSIZE*4]);

    tmp0 = (z2 + z3) << CONST_BITS;
    tmp1 = (z2 - z3) << CONST_BITS;

    tmp10 = tmp0 + tmp3;
    tmp13 = tmp0 - tmp3;
    tmp11 = tmp1 + tmp2;
    tmp12 = tmp1 - tmp2;

    /* Odd part per figure 8; the matrix is unitary and hence its
     * transpose is its inverse. i0..i3 are y7,y5,y3,y1 respectively.
     */

    tmp0 = DEQUANTIZE(inptr[DCTSIZE*7], quantptr[DCTSIZE*7]);
    tmp1 = DEQUANTIZE(inptr[DCTSIZE*5], quantptr[DCTSIZE*5]);
    tmp2 = DEQUANTIZE(inptr[DCTSIZE*3], quantptr[DCTSIZE*3]);
    tmp3 = DEQUANTIZE(inptr[DCTSIZE*1], quantptr[DCTSIZE*1]);

    z1 = tmp0 + tmp3;
    z2 = tmp1 + tmp2;
    z3 = tmp0 + tmp2;
    z4 = tmp1 + tmp3;
    z5 = MULTIPLY(z3 + z4, FIX_1_175875602); /* sqrt(2) * c3 */

    tmp0 = MULTIPLY(tmp0, FIX_0_298631336); /* sqrt(2) * (-c1+c3+c5-c7) */
    tmp1 = MULTIPLY(tmp1, FIX_2_053119869); /* sqrt(2) * (c1+c3-c5+c7) */

```

```

tmp2 = MULTIPLY(tmp2, FIX_072711026); /* sqrt(2) * (c1+c3+c5+c7) */
tmp3 = MULTIPLY(tmp3, FIX_501321110); /* sqrt(2) * (c1+c3-c5-c7) */
z1 = MULTIPLY(z1, -FIX_009976223); /* sqrt(2) * (c7-c3) */
z2 = MULTIPLY(z2, -FIX_2_562915447); /* sqrt(2) * (-c1-c3) */
z3 = MULTIPLY(z3, -FIX_1_961570560); /* sqrt(2) * (-c3-c5) */
z4 = MULTIPLY(z4, -FIX_0_390180644); /* sqrt(2) * (c5-c3) */

z3 += z5;
z4 += z5;

tmp0 += z1 + z3;
tmp1 += z2 + z4;
tmp2 += z2 + z3;
tmp3 += z1 + z4;

/* Final output stage: inputs are tmp10..tmp13, tmp0..tmp3 */

wsptr[DCTSIZE*0] = (int) DESCALE(tmp10 + tmp3, CONST_BITS-PASS1_BITS);
wsptr[DCTSIZE*7] = (int) DESCALE(tmp10 - tmp3, CONST_BITS-PASS1_BITS);
wsptr[DCTSIZE*1] = (int) DESCALE(tmp11 + tmp2, CONST_BITS-PASS1_BITS);
wsptr[DCTSIZE*6] = (int) DESCALE(tmp11 - tmp2, CONST_BITS-PASS1_BITS);
wsptr[DCTSIZE*2] = (int) DESCALE(tmp12 + tmp1, CONST_BITS-PASS1_BITS);
wsptr[DCTSIZE*5] = (int) DESCALE(tmp12 - tmp1, CONST_BITS-PASS1_BITS);
wsptr[DCTSIZE*3] = (int) DESCALE(tmp13 + tmp0, CONST_BITS-PASS1_BITS);
wsptr[DCTSIZE*4] = (int) DESCALE(tmp13 - tmp0, CONST_BITS-PASS1_BITS);

inptr++; /* advance pointers to next column */
quantptr++;
wsptr++;
)

/* Pass 2: process rows from work array, store into output array. */
/* Note that we must descale the results by a factor of 8 == 2**3, */
/* and also undo the PASS1_BITS scaling. */
wsptr = workspace;
for (ctr = 0; ctr < DCTSIZE; ctr++) {
    outptr = output_buf[ctr] + output_col;
    /* Rows of zeroes can be exploited in the same way as we did with columns.
     * However, the column calculation has created many nonzero AC terms, so
     * the simplification applies less often (typically 5% to 10% of the time).
     * On machines with very fast multiplication, it's possible that the
     * test takes more time than it's worth. In that case this section
     * may be commented out.
     */
    #ifndef NO_ZERO_ROW_TEST
    if (wsptr[1] == 0 && wsptr[2] == 0 && wsptr[3] == 0 && wsptr[4] == 0 &&
        wsptr[5] == 0 && wsptr[6] == 0 && wsptr[7] == 0) {
        /* AC terms all zero */
        JSAMPLE dval = range_limit[(int) DESCALE((INT32) wsptr[0], PASS1_BITS+3)
            & RANGE_MASK];
        outptr[0] = dval;
        outptr[1] = dval;
        outptr[2] = dval;
        outptr[3] = dval;
        outptr[4] = dval;
        outptr[5] = dval;
        outptr[6] = dval;
        outptr[7] = dval;

        wsptr += DCTSIZE; /* advance pointer to next row */
        continue;
    }
    #endif

    /* Even part: reverse the even part of the forward DCT. */
    /* The rotator is sqrt(2)*c(-6). */

    z2 = (INT32) wsptr[2];
    z3 = (INT32) wsptr[6];

    z1 = MULTIPLY(z2 + z3, FIX_0_541196100);
    tmp2 = z1 + MULTIPLY(z3, -FIX_1_847759065);
    tmp3 = z1 + MULTIPLY(z2, FIX_0_765366865);

    tmp0 = ((INT32) wsptr[0] + (INT32) wsptr[4]) << CONST_BITS;
    tmp1 = ((INT32) wsptr[0] - (INT32) wsptr[4]) << CONST_BITS;

```

```

tmp10 = tmp0 + tmp3;
tmp13 = tmp0 - tmp3;
tmp11 = tmp1 + tmp2;
tmp12 = tmp1 - tmp2;

/* Odd part per figure 8; the matrix is unitary and hence its
 * transpose is its inverse. i0..i3 are y7,y5,y3,y1 respectively.
 */

tmp0 = (INT32) wsptr[7];
tmp1 = (INT32) wsptr[5];
tmp2 = (INT32) wsptr[3];
tmp3 = (INT32) wsptr[1];

z1 = tmp0 + tmp3;
z2 = tmp1 + tmp2;
z3 = tmp0 + tmp2;
z4 = tmp1 + tmp3;
z5 = MULTIPLY(z3 + z4, FIX_1_175875602); /* sqrt(2) * c3 */

tmp0 = MULTIPLY(tmp0, FIX_0_298631336); /* sqrt(2) * (-c1+c3+c5-c7) */
tmp1 = MULTIPLY(tmp1, FIX_2_053119869); /* sqrt(2) * ( c1+c3-c5+c7) */
tmp2 = MULTIPLY(tmp2, FIX_3_072711026); /* sqrt(2) * ( c1+c3+c5-c7) */
tmp3 = MULTIPLY(tmp3, FIX_1_501321110); /* sqrt(2) * ( c1+c3-c5-c7) */
z1 = MULTIPLY(z1, - FIX_0_899976223); /* sqrt(2) * (c7-c3) */
z2 = MULTIPLY(z2, - FIX_2_562915447); /* sqrt(2) * (-c1-c3) */
z3 = MULTIPLY(z3, - FIX_1_961570560); /* sqrt(2) * (-c3-c5) */
z4 = MULTIPLY(z4, - FIX_0_390180644); /* sqrt(2) * (c5-c3) */

z3 += z5;
z4 += z5;

tmp0 += z1 + z3;
tmp1 += z2 + z4;
tmp2 += z2 + z3;
tmp3 += z1 + z4;

/* Final output stage: inputs are tmp10..tmp13, tmp0..tmp3 */

outptr[0] = range_limit[(int) DESCALE(tmp10 + tmp3,
CONST_BITS+PASS1_BITS+3)
& RANGE_MASK];
outptr[7] = range_limit[(int) DESCALE(tmp10 - tmp3,
CONST_BITS+PASS1_BITS+3)
& RANGE_MASK];
outptr[1] = range_limit[(int) DESCALE(tmp11 + tmp2,
CONST_BITS+PASS1_BITS+3)
& RANGE_MASK];
outptr[6] = range_limit[(int) DESCALE(tmp11 - tmp2,
CONST_BITS+PASS1_BITS+3)
& RANGE_MASK];
outptr[2] = range_limit[(int) DESCALE(tmp12 + tmp1,
CONST_BITS+PASS1_BITS+3)
& RANGE_MASK];
outptr[5] = range_limit[(int) DESCALE(tmp12 - tmp1,
CONST_BITS+PASS1_BITS+3)
& RANGE_MASK];
outptr[3] = range_limit[(int) DESCALE(tmp13 + tmp0,
CONST_BITS+PASS1_BITS+3)
& RANGE_MASK];
outptr[4] = range_limit[(int) DESCALE(tmp13 - tmp0,
CONST_BITS+PASS1_BITS+3)
& RANGE_MASK];

wsptr += DCTSIZE; /* advance pointer to next row */
}
}
#endif /* DCT_ISLOW_SUPPORTED */

```

```

/*
 * jidctred.c
 *
 * Copyright (C) 1994-1998, Thomas G. Lane.
 * This file is part of the Independent JPEG Group's software.
 * For conditions of distribution and use, see the accompanying README file.
 *
 * This file contains inverse-DCT routines that produce reduced-size output:
 * either 4x4, 2x2, or 1x1 pixels from an 8x8 DCT block.
 *
 * The implementation is based on the Loeffler, Ligtenberg and Moschytz (LL&M)
 * algorithm used in jidctint.c. We simply replace each 8-to-8 1-D IDCT step
 * with an 8-to-4 step that produces the four averages of two adjacent outputs
 * (or an 8-to-2 step producing two averages of four outputs, for 2x2 output).
 * These steps were derived by computing the corresponding values at the end
 * of the normal LL&M code, then simplifying as much as possible.
 *
 * 1x1 is trivial: just take the DC coefficient divided by 8.
 *
 * See jidctint.c for additional comments.
 */

```

```

#define JPEG_INTERNALS
#include "jinclude.h"
#include "jpeglib.h"
#include "jdct.h"      /* Private declarations for DCT subsystem */

#ifdef IDCT_SCALING_SUPPORTED

```

```

/*
 * This module is specialized to the case DCTSIZE = 8.
 */
#ifdef DCTSIZE != 8
  Sorry, this code only copes with 8x8 DCTs. /* deliberate syntax err */
#endif

/* Scaling is the same as in jidctint.c. */
#ifdef BITS_IN_JSAMPLE == 8
#define CONST_BITS 13
#define PASS1_BITS 2
#else
#define CONST_BITS 13
#define PASS1_BITS 1      /* lose a little precision to avoid overflow */
#endif

/* Some C compilers fail to reduce "FIX(constant)" at compile time, thus
 * causing a lot of useless floating-point operations at run time.
 * To get around this we use the following pre-calculated constants.
 * If you change CONST_BITS you may want to add appropriate values.
 * (With a reasonable C compiler, you can just rely on the FIX() macro...)
 */

```

```

#ifdef CONST_BITS == 13
#define FIX_0_211164243 ((INT32) 1730) /* FIX(0.211164243) */
#define FIX_0_509795579 ((INT32) 4176) /* FIX(0.509795579) */
#define FIX_0_601344887 ((INT32) 4926) /* FIX(0.601344887) */
#define FIX_0_720959822 ((INT32) 5906) /* FIX(0.720959822) */
#define FIX_0_765366865 ((INT32) 6270) /* FIX(0.765366865) */
#define FIX_0_850430095 ((INT32) 6967) /* FIX(0.850430095) */
#define FIX_0_899976223 ((INT32) 7373) /* FIX(0.899976223) */
#define FIX_1_061594337 ((INT32) 8697) /* FIX(1.061594337) */
#define FIX_1_272758580 ((INT32) 10426) /* FIX(1.272758580) */
#define FIX_1_451774981 ((INT32) 11893) /* FIX(1.451774981) */
#define FIX_1_847759065 ((INT32) 15137) /* FIX(1.847759065) */
#define FIX_2_172734803 ((INT32) 17799) /* FIX(2.172734803) */
#define FIX_2_562915447 ((INT32) 20995) /* FIX(2.562915447) */
#define FIX_3_624509785 ((INT32) 29692) /* FIX(3.624509785) */
#else
#define FIX_0_211164243 FIX(0.211164243)
#define FIX_0_509795579 FIX(0.509795579)
#define FIX_0_601344887 FIX(0.601344887)
#define FIX_0_720959822 FIX(0.720959822)
#define FIX_0_765366865 FIX(0.765366865)
#define FIX_0_850430095 FIX(0.850430095)
#define FIX_0_899976223 FIX(0.899976223)
#define FIX_1_061594337 FIX(1.061594337)

```



```

#define FIX_1_272758580 FIX(1.272758580)
#define FIX_1_451774981 FIX(1.451774981)
#define FIX_1_847759065 FIX(1.847759065)
#define FIX_2_172734803 FIX(2.172734803)
#define FIX_2_562915447 FIX(2.562915447)
#define FIX_3_624509785 FIX(3.624509785)
#endif

/* Multiply an INT32 variable by an INT32 constant to yield an INT32 result.
 * For 8-bit samples with the recommended scaling, all the variable
 * and constant values involved are no more than 16 bits wide, so a
 * 16x16->32 bit multiply can be used instead of a full 32x32 multiply.
 * For 12-bit samples, a full 32-bit multiplication will be needed.
 */

#if BITS_IN_JSAMPLE == 8
#define MULTIPLY(var,const) MULTIPLY16C16(var,const)
#else
#define MULTIPLY(var,const) ((var) * (const))
#endif

/* Dequantize a coefficient by multiplying it by the multiplier-table
 * entry; produce an int result. In this module, both inputs and result
 * are 16 bits or less, so either int or short multiply will work.
 */

#define DEQUANTIZE(coef,quantval) (((ISLOW_MULT_TYPE) (coef)) * (quantval))

/*
 * Perform dequantization and inverse DCT on one block of coefficients,
 * producing a reduced-size 4x4 output block.
 */
GLOBAL(void)
jpeg_idct_4x4 (j_decompress_ptr cinfo, jpeg_component_info * comp_ptr,
               JCOEFPTR coef_block,
               JSAMPARRAY output_buf, JDIMENSION output_col)
{
    INT32 tmp0, tmp2, tmp10, tmp12;
    INT32 z1, z2, z3, z4;
    JCOEFPTR inptr;
    ISLOW_MULT_TYPE * quantptr;
    int * wsptr;
    JSAMPROW outptr;
    JSAMPLE *range_limit = IDCT_range_limit(cinfo);
    int ctr;
    int workspace[DCTSIZE*4]; /* buffers data between passes */
    SHIFT_TEMPS

    /* Pass 1: process columns from input, store into work array. */

    inptr = coef_block;
    quantptr = (ISLOW_MULT_TYPE *) comp_ptr-> dct_table;
    wsptr = workspace;
    for (ctr = DCTSIZE; ctr > 0; inptr++, quantptr++, wsptr++, ctr--) {
        /* Don't bother to process column 4, because second pass won't use it */
        if (ctr == DCTSIZE-4)
            continue;
        if (inptr[DCTSIZE*1] == 0 && inptr[DCTSIZE*2] == 0 &&
            inptr[DCTSIZE*3] == 0 && inptr[DCTSIZE*5] == 0 &&
            inptr[DCTSIZE*6] == 0 && inptr[DCTSIZE*7] == 0) {
            /* AC terms all zero; we need not examine term 4 for 4x4 output */
            int dcv = DEQUANTIZE(inptr[DCTSIZE*0], quantptr[DCTSIZE*0]) << PASS1_BITS;

            wsptr[DCTSIZE*0] = dcv;
            wsptr[DCTSIZE*1] = dcv;
            wsptr[DCTSIZE*2] = dcv;
            wsptr[DCTSIZE*3] = dcv;

            continue;
        }

        /* Even part */

        tmp0 = DEQUANTIZE(inptr[DCTSIZE*0], quantptr[DCTSIZE*0]);
        tmp0 <<= (CONST_BITS+1);

```

```

z2 = DEQUANTIZE(inpbuf[DCTSIZE*2], quantptr[DCTSIZE*2]);
z3 = DEQUANTIZE(inpbuf[DCTSIZE*6], quantptr[DCTSIZE*6]);

tmp2 = MULTIPLY(z2, FIX_1_847759065) + MULTIPLY(z3, - FIX_0_765366865);

tmp10 = tmp0 + tmp2;
tmp12 = tmp0 - tmp2;

/* Odd part */

z1 = DEQUANTIZE(inpbuf[DCTSIZE*7], quantptr[DCTSIZE*7]);
z2 = DEQUANTIZE(inpbuf[DCTSIZE*5], quantptr[DCTSIZE*5]);
z3 = DEQUANTIZE(inpbuf[DCTSIZE*3], quantptr[DCTSIZE*3]);
z4 = DEQUANTIZE(inpbuf[DCTSIZE*1], quantptr[DCTSIZE*1]);

tmp0 = MULTIPLY(z1, - FIX_0_211164243) /* sqrt(2) * (c3-c1) */
+ MULTIPLY(z2, FIX_1_451774981) /* sqrt(2) * (c3+c7) */
+ MULTIPLY(z3, - FIX_2_172734803) /* sqrt(2) * (-c1-c5) */
+ MULTIPLY(z4, FIX_1_061594337); /* sqrt(2) * (c5+c7) */

tmp2 = MULTIPLY(z1, - FIX_0_509795579) /* sqrt(2) * (c7-c5) */
+ MULTIPLY(z2, - FIX_0_601344887) /* sqrt(2) * (c5-c1) */
+ MULTIPLY(z3, FIX_0_899976223) /* sqrt(2) * (c3-c7) */
+ MULTIPLY(z4, FIX_2_562915447); /* sqrt(2) * (c1+c3) */

/* Final output stage */

wsptr[DCTSIZE*0] = (int) DESCALE(tmp10 + tmp2, CONST_BITS-PASS1_BITS+1);
wsptr[DCTSIZE*3] = (int) DESCALE(tmp10 - tmp2, CONST_BITS-PASS1_BITS+1);
wsptr[DCTSIZE*1] = (int) DESCALE(tmp12 + tmp0, CONST_BITS-PASS1_BITS+1);
wsptr[DCTSIZE*2] = (int) DESCALE(tmp12 - tmp0, CONST_BITS-PASS1_BITS+1);

/* Pass 2: process 4 rows from work array, store into output array. */
wsptr = workspace;
for (ctr = 0; ctr < 4; ctr++) {
    outptr = output_buf[ctr] + output_col;
    /* It's not clear whether a zero row test is worthwhile here ... */

#ifdef NO_ZERO_ROW_TEST
    if (wsptr[1] == 0 && wsptr[2] == 0 && wsptr[3] == 0 &&
        wsptr[5] == 0 && wsptr[6] == 0 && wsptr[7] == 0) {
        /* AC terms all zero */
        JSAMPLE dcvall = range_limit[(int) DESCALE((INT32) wsptr[0], PASS1_BITS+3)
            & RANGE_MASK];

        outptr[0] = dcvall;
        outptr[1] = dcvall;
        outptr[2] = dcvall;
        outptr[3] = dcvall;

        wsptr += DCTSIZE; /* advance pointer to next row */
        continue;
    }
#endif

/* Even part */

tmp0 = ((INT32) wsptr[0]) << (CONST_BITS+1);

tmp2 = MULTIPLY((INT32) wsptr[2], FIX_1_847759065)
+ MULTIPLY((INT32) wsptr[6], - FIX_0_765366865);

tmp10 = tmp0 + tmp2;
tmp12 = tmp0 - tmp2;

/* Odd part */

z1 = (INT32) wsptr[7];
z2 = (INT32) wsptr[5];
z3 = (INT32) wsptr[3];
z4 = (INT32) wsptr[1];

tmp0 = MULTIPLY(z1, - FIX_0_211164243) /* sqrt(2) * (c3-c1) */
+ MULTIPLY(z2, FIX_1_451774981) /* sqrt(2) * (c3+c7) */
+ MULTIPLY(z3, - FIX_2_172734803) /* sqrt(2) * (-c1-c5) */
+ MULTIPLY(z4, FIX_1_061594337); /* sqrt(2) * (c5+c7) */

tmp2 = MULTIPLY(z1, - FIX_0_509795579) /* sqrt(2) * (c7-c5) */

```

```

+ MULTIPLY(z2, - FIX_0_601344887) /* sqrt(2) * (c5-c1) */
+ MULTIPLY(z3, FIX_0_896223) /* sqrt(2) * (c3-c7) */
+ MULTIPLY(z4, FIX_2_5626447); /* sqrt(2) * (c1+c3) */

/* Final output stage */

outptr[0] = range_limit[(int) DESCALE(tmp10 + tmp2,
CONST_BITS+PASS1_BITS+3+1)
& RANGE_MASK];
outptr[3] = range_limit[(int) DESCALE(tmp10 - tmp2,
CONST_BITS+PASS1_BITS+3+1)
& RANGE_MASK];
outptr[1] = range_limit[(int) DESCALE(tmp12 + tmp0,
CONST_BITS+PASS1_BITS+3+1)
& RANGE_MASK];
outptr[2] = range_limit[(int) DESCALE(tmp12 - tmp0,
CONST_BITS+PASS1_BITS+3+1)
& RANGE_MASK];

wsptr += DCTSIZE; /* advance pointer to next row */
}

/*
* Perform dequantization and inverse DCT on one block of coefficients,
* producing a reduced-size 2x2 output block.
*/

GLOBAL(void)
jpeg_idct_2x2 (j_decompress_ptr cinfo, jpeg_component_info * compptr,
JCOEFPTR coef_block,
JSAMPARRAY output_buf, JDIMENSION output_col)
{
    INT32 tmp0, tmp10, z1;
    JCOEFPTR inptr;
    ISLOW_MULT_TYPE * quantptr;
    int * wsptr;
    JSAMPROW outptr;
    JSAMPLE *range_limit = IDCT_range_limit(cinfo);
    int ctr;
    int workspace[DCTSIZE*2]; /* buffers data between passes */
    SHIFT_TEMPS

    /* Pass 1: process columns from input, store into work array. */
    inptr = coef_block;
    quantptr = (ISLOW_MULT_TYPE *) compptr->dct_table;
    wsptr = workspace;
    for (ctr = DCTSIZE; ctr > 0; inptr++, quantptr++, wsptr++, ctr--) {
        /* Don't bother to process columns 2,4,6 */
        if (ctr == DCTSIZE-2 || ctr == DCTSIZE-4 || ctr == DCTSIZE-6)
            continue;
        if (inptr[DCTSIZE*1] == 0 && inptr[DCTSIZE*3] == 0 &&
            inptr[DCTSIZE*5] == 0 && inptr[DCTSIZE*7] == 0) {
            /* AC terms all zero; we need not examine terms 2,4,6 for 2x2 output */
            int dcval = DEQUANTIZE(inptr[DCTSIZE*0], quantptr[DCTSIZE*0]) << PASS1_BITS;

            wsptr[DCTSIZE*0] = dcval;
            wsptr[DCTSIZE*1] = dcval;

            continue;
        }

        /* Even part */

        z1 = DEQUANTIZE(inptr[DCTSIZE*0], quantptr[DCTSIZE*0]);
        tmp10 = z1 << (CONST_BITS+2);

        /* Odd part */

        z1 = DEQUANTIZE(inptr[DCTSIZE*7], quantptr[DCTSIZE*7]);
        tmp0 = MULTIPLY(z1, - FIX_0_720959822); /* sqrt(2) * (c7-c5+c3-c1) */
        z1 = DEQUANTIZE(inptr[DCTSIZE*5], quantptr[DCTSIZE*5]);
        tmp0 += MULTIPLY(z1, FIX_0_850430095); /* sqrt(2) * (-c1+c3+c5+c7) */
        z1 = DEQUANTIZE(inptr[DCTSIZE*3], quantptr[DCTSIZE*3]);
        tmp0 += MULTIPLY(z1, - FIX_1_272758580); /* sqrt(2) * (-c1+c3-c5-c7) */
        z1 = DEQUANTIZE(inptr[DCTSIZE*1], quantptr[DCTSIZE*1]);
        tmp0 += MULTIPLY(z1, FIX_3_624509785); /* sqrt(2) * (c1+c3+c5+c7) */
    }

```

```

/* Final output stage */
wsptr[DCTSIZE*0] = (int) DESCALE(tmp10 + tmp0, CONST_BITS-PASS1_BITS+2);
wsptr[DCTSIZE*1] = (int) DESCALE(tmp10 - tmp0, CONST_BITS-PASS1_BITS+2);
}

/* Pass 2: process 2 rows from work array, store into output array. */
wsptr = workspace;
for (ctr = 0; ctr < 2; ctr++) {
    outptr = output_buf[ctr] + output_col;
    /* It's not clear whether a zero row test is worthwhile here ... */

#ifdef NO_ZERO_ROW_TEST
    if (wsptr[1] == 0 && wsptr[3] == 0 && wsptr[5] == 0 && wsptr[7] == 0) {
        /* AC terms all zero */
        JSAMPLE dval = range_limit[(int) DESCALE((INT32) wsptr[0], PASS1_BITS+3)
            & RANGE_MASK];

        outptr[0] = dval;
        outptr[1] = dval;

        wsptr += DCTSIZE;      /* advance pointer to next row */
        continue;
    }
#endif

    /* Even part */

    tmp10 = ((INT32) wsptr[0]) << (CONST_BITS+2);

    /* Odd part */

    tmp0 = MULTIPLY((INT32) wsptr[7], - FIX_0_720959822) /* sqrt(2) * (c7-c5+c3-c1) */
        + MULTIPLY((INT32) wsptr[5], FIX_0_850430095) /* sqrt(2) * (-c1+c3+c5+c7) */
        + MULTIPLY((INT32) wsptr[3], - FIX_1_272758580) /* sqrt(2) * (-c1+c3-c5-c7) */
        + MULTIPLY((INT32) wsptr[1], FIX_3_624509785); /* sqrt(2) * (c1+c3+c5+c7) */

    /* Final output stage */

    outptr[0] = range_limit[(int) DESCALE(tmp10 + tmp0,
        CONST_BITS+PASS1_BITS+3+2)
        & RANGE_MASK];
    outptr[1] = range_limit[(int) DESCALE(tmp10 - tmp0,
        CONST_BITS+PASS1_BITS+3+2)
        & RANGE_MASK];

    wsptr += DCTSIZE;      /* advance pointer to next row */

    /*
     * Perform dequantization and inverse DCT on one block of coefficients,
     * producing a reduced-size 1x1 output block.
     */
}

GLOBAL(void)
jpeg_idct_1x1 (j_decompress_ptr cinfo, jpeg_component_info * comp_ptr,
    JCOEFPTR coef_block,
    JSAMPARRAY output_buf, JDIMENSION output_col)
{
    int dval;
    ISLOW_MULT_TYPE * quant_ptr;
    JSAMPLE *range_limit = IDCT_range_limit(cinfo);
    SHIFT_TEMPS

    /* We hardly need an inverse DCT routine for this: just take the
     * average pixel value, which is one-eighth of the DC coefficient.
     */
    quant_ptr = (ISLOW_MULT_TYPE *) comp_ptr->dct_table;
    dval = DEQUANTIZE(coef_block[0], quant_ptr[0]);
    dval = (int) DESCALE((INT32) dval, 3);

    output_buf[0][output_col] = range_limit[dval & RANGE_MASK];
}

#endif /* IDCT_SCALING_SUPPORTED */

```

```

/*
 * jmemmgr.c
 *
 * Copyright (C) 1991-1997, Thomas G. Lane.
 * This file is part of the Independent JPEG Group's software.
 * For conditions of distribution and use, see the accompanying README file.
 *
 * This file contains the JPEG system-independent memory management
 * routines. This code is usable across a wide variety of machines; most
 * of the system dependencies have been isolated in a separate file.
 * The major functions provided here are:
 *   * pool-based allocation and freeing of memory;
 *   * policy decisions about how to divide available memory among the
 *     virtual arrays;
 *   * control logic for swapping virtual arrays between main memory and
 *     backing storage.
 * The separate system-dependent file provides the actual backing-storage
 * access code, and it contains the policy decision about how much total
 * main memory to use.
 * This file is system-dependent in the sense that some of its functions
 * are unnecessary in some systems. For example, if there is enough virtual
 * memory so that backing storage will never be used, much of the virtual
 * array control logic could be removed. (Of course, if you have that much
 * memory then you shouldn't care about a little bit of unused code...)
 */

```

```

#define JPEG_INTERNALS
#define AM_MEMORY_MANAGER /* we define jvirt_xarray_control structs */
#include "jinclude.h"
#include "jpeglib.h"
#include "jmemsys.h" /* import the system-dependent declarations */

#ifdef NO_GETENV
#ifndef HAVE_STDLIB_H /* <stdlib.h> should declare getenv() */
extern char * getenv JPP((const char * name));
#endif
#endif

```

#### Some important notes:

The allocation routines provided here must never return NULL.  
They should exit to error\_exit if unsuccessful.

It's not a good idea to try to merge the sarray and barray routines,  
even though they are textually almost the same, because samples are  
usually stored as bytes while coefficients are shorts or ints. Thus,  
in machines where byte pointers have a different representation from  
word pointers, the resulting machine code could not be the same.

```

/*
 * Many machines require storage alignment: longs must start on 4-byte
 * boundaries, doubles on 8-byte boundaries, etc. On such machines, malloc()
 * always returns pointers that are multiples of the worst-case alignment
 * requirement, and we had better do so too.
 * There isn't any really portable way to determine the worst-case alignment
 * requirement. This module assumes that the alignment requirement is
 * multiples of sizeof(ALIGN_TYPE).
 * By default, we define ALIGN_TYPE as double. This is necessary on some
 * workstations (where doubles really do need 8-byte alignment) and will work
 * fine on nearly everything. If your machine has lesser alignment needs,
 * you can save a few bytes by making ALIGN_TYPE smaller.
 * The only place I know of where this will NOT work is certain Macintosh
 * 680x0 compilers that define double as a 10-byte IEEE extended float.
 * Doing 10-byte alignment is counterproductive because longwords won't be
 * aligned well. Put "#define ALIGN_TYPE long" in jconfig.h if you have
 * such a compiler.
 */

```

```

#ifdef ALIGN_TYPE /* so can override from jconfig.h */
#define ALIGN_TYPE double
#endif

```

```

/*
 * We allocate objects from "pools", where each pool is gotten with a single
 * request to jpeg_get_small() or jpeg_get_large(). There is no per-object
 * overhead within a pool, except for alignment padding. Each pool has a

```

```

* header with a link to the next pool of the same class.
* Small and large pool headers are identical except that the latter
* link pointer must be FAR on 80x86 machines.
* Notice that the "real" header fields are union'ed with a dummy ALIGN_TYPE
* field. This forces the compiler to make SIZEOF(small_pool_hdr) a multiple
* of the alignment requirement of ALIGN_TYPE.
*/

```

```
typedef union small_pool_struct * small_pool_ptr;
```

```
typedef union small_pool_struct {
    struct {
        small_pool_ptr next;    /* next in list of pools */
        size_t bytes_used;      /* how many bytes already used within pool */
        size_t bytes_left;      /* bytes still available in this pool */
    } hdr;
    ALIGN_TYPE dummy;          /* included in union to ensure alignment */
} small_pool_hdr;
```

```
typedef union large_pool_struct * large_pool_ptr;
```

```
typedef union large_pool_struct {
    struct {
        large_pool_ptr next;    /* next in list of pools */
        size_t bytes_used;      /* how many bytes already used within pool */
        size_t bytes_left;      /* bytes still available in this pool */
    } hdr;
    ALIGN_TYPE dummy;          /* included in union to ensure alignment */
} large_pool_hdr;
```

```
/*
 * Here is the full definition of a memory manager object.
 */

```

```
typedef struct {
    struct jpeg_memory_mgr pub;    /* public fields */

    /* Each pool identifier (lifetime class) names a linked list of pools. */
    small_pool_ptr small_list[JPOOL_NUMPOOLS];
    large_pool_ptr large_list[JPOOL_NUMPOOLS];

    /* Since we only have one lifetime class of virtual arrays, only one
     * linked list is necessary (for each datatype). Note that the virtual
     * array control blocks being linked together are actually stored somewhere
     * in the small-pool list.
     */
    jvirt_sarray_ptr virt_sarray_list;
    jvirt_barray_ptr virt_barray_list;

    /* This counts total space obtained from jpeg_get_small/large */
    long total_space_allocated;

    /* alloc_sarray and alloc_barray set this value for use by virtual
     * array routines.
     */
    JDIMENSION last_rowsperchunk; /* from most recent alloc_sarray/barray */
} my_memory_mgr;
```

```
typedef my_memory_mgr * my_mem_ptr;
```

```
/*
 * The control blocks for virtual arrays.
 * Note that these blocks are allocated in the "small" pool area.
 * System-dependent info for the associated backing store (if any) is hidden
 * inside the backing_store_info struct.
 */

```

```
struct jvirt_sarray_control {
    JSAMPARRAY mem_buffer;    /* => the in-memory buffer */
    JDIMENSION rows_in_array; /* total virtual array height */
    JDIMENSION samplesperrow; /* width of array (and of memory buffer) */
    JDIMENSION maxaccess;     /* max rows accessed by access_virt_sarray */
    JDIMENSION rows_in_mem;    /* height of memory buffer */
    JDIMENSION rowsperchunk;   /* allocation chunk size in mem_buffer */
    JDIMENSION cur_start_row;  /* first logical row # in the buffer */
    JDIMENSION first_undef_row; /* row # of first uninitialized row */
    boolean pre_zero;          /* pre-zero mode requested? */
    boolean dirty;             /* do current buffer contents need written? */
};

```

```

boolean b_s_open; /* is backing-store data valid? */
jvirt_sarray_ptr next; /* link to next virtual sarray control block */
backing_store_info b_s_info; /* System-dependent control info */
};

struct jvirt_barray_control {
    JBLOCKARRAY mem_buffer; /* => the in-memory buffer */
    JDIMENSION rows_in_array; /* total virtual array height */
    JDIMENSION blocksperrrow; /* width of array (and of memory buffer) */
    JDIMENSION maxaccess; /* max rows accessed by access_virt_barray */
    JDIMENSION rows_in_mem; /* height of memory buffer */
    JDIMENSION rowsperchunk; /* allocation chunk size in mem_buffer */
    JDIMENSION cur_start_row; /* first logical row # in the buffer */
    JDIMENSION first_undef_row; /* row # of first uninitialized row */
    boolean pre_zero; /* pre-zero mode requested? */
    boolean dirty; /* do current buffer contents need written? */
    boolean b_s_open; /* is backing-store data valid? */
    jvirt_barray_ptr next; /* link to next virtual barray control block */
    backing_store_info b_s_info; /* System-dependent control info */
};

#ifdef MEM_STATS /* optional extra stuff for statistics */

LOCAL(void)
print_mem_stats (j_common_ptr cinfo, int pool_id)
{
    my_mem_ptr mem = (my_mem_ptr) cinfo->mem;
    small_pool_ptr shdr_ptr;
    large_pool_ptr lhdr_ptr;

    /* Since this is only a debugging stub, we can cheat a little by using
     * fprintf directly rather than going through the trace message code.
     * This is helpful because message parm array can't handle longs.
     */
    fprintf(stderr, "Freeing pool %d, total space = %ld\n",
        pool_id, mem->total_space_allocated);

    for (lhdr_ptr = mem->large_list[pool_id]; lhdr_ptr != NULL;
        lhdr_ptr = lhdr_ptr->hdr.next) {
        fprintf(stderr, " Large chunk used %ld\n",
            (long) lhdr_ptr->hdr.bytes_used);
    }

    for (shdr_ptr = mem->small_list[pool_id]; shdr_ptr != NULL;
        shdr_ptr = shdr_ptr->hdr.next) {
        fprintf(stderr, " Small chunk used %ld free %ld\n",
            (long) shdr_ptr->hdr.bytes_used,
            (long) shdr_ptr->hdr.bytes_left);
    }
}

#endif /* MEM_STATS */

LOCAL(void)
out_of_memory (j_common_ptr cinfo, int which)
/* Report an out-of-memory error and stop execution */
/* If we compiled MEM_STATS support, report alloc requests before dying */
{
#ifdef MEM_STATS
    cinfo->err->trace_level = 2; /* force self_destruct to report stats */
#endif
    ERREXIT1(cinfo, JERR_OUT_OF_MEMORY, which);
}

/*
 * Allocation of "small" objects.
 *
 * For these, we use pooled storage. When a new pool must be created,
 * we try to get enough space for the current request plus a "slop" factor,
 * where the slop will be the amount of leftover space in the new pool.
 * The speed vs. space tradeoff is largely determined by the slop values.
 * A different slop value is provided for each pool class (lifetime),
 * and we also distinguish the first pool of a class from later ones.
 * NOTE: the values given work fairly well on both 16- and 32-bit-int
 * machines, but may be too small if longs are 64 bits or more.
 */

```

```

static const size_t first_pool_slop[JPOOL_NUMPOOLS] =
{
    1600,          /* first PERMANENT pool */
    16000         /* first IMAGE pool */
};

static const size_t extra_pool_slop[JPOOL_NUMPOOLS] =
{
    0,            /* additional PERMANENT pools */
    5000         /* additional IMAGE pools */
};

#define MIN_SLOP 50          /* greater than 0 to avoid futile looping */

METHODDEF(void *)
alloc_small (j_common_ptr cinfo, int pool_id, size_t sizeobject)
/* Allocate a "small" object */
{
    my_mem_ptr mem = (my_mem_ptr) cinfo->mem;
    small_pool_ptr hdr_ptr, prev_hdr_ptr;
    char * data_ptr;
    size_t odd_bytes, min_request, slop;

    /* Check for unsatisfiable request (do now to ensure no overflow below) */
    if (sizeobject > (size_t) (MAX_ALLOC_CHUNK-SIZEOF(small_pool_hdr)))
        out_of_memory(cinfo, 1); /* request exceeds malloc's ability */

    /* Round up the requested size to a multiple of SIZEOF(ALIGN_TYPE) */
    odd_bytes = sizeobject % SIZEOF(ALIGN_TYPE);
    if (odd_bytes > 0)
        sizeobject += SIZEOF(ALIGN_TYPE) - odd_bytes;

    /* See if space is available in any existing pool */
    if (pool_id < 0 || pool_id >= JPOOL_NUMPOOLS)
        ERREXIT1(cinfo, JERR_BAD_POOL_ID, pool_id); /* safety check */
    prev_hdr_ptr = NULL;
    hdr_ptr = mem->small_list[pool_id];
    while (hdr_ptr != NULL) {
        if (hdr_ptr->hdr.bytes_left >= sizeobject)
            break; /* found pool with enough space */
        prev_hdr_ptr = hdr_ptr;
        hdr_ptr = hdr_ptr->hdr.next;
    }

    /* Time to make a new pool? */
    if (hdr_ptr == NULL) {
        /* min_request is what we need now, slop is what will be leftover */
        min_request = sizeobject + SIZEOF(small_pool_hdr);
        if (prev_hdr_ptr == NULL) /* first pool in class? */
            slop = first_pool_slop[pool_id];
        else
            slop = extra_pool_slop[pool_id];
        /* Don't ask for more than MAX_ALLOC_CHUNK */
        if (slop > (size_t) (MAX_ALLOC_CHUNK-min_request))
            slop = (size_t) (MAX_ALLOC_CHUNK-min_request);
        /* Try to get space, if fail reduce slop and try again */
        for (;;) {
            hdr_ptr = (small_pool_ptr) jpeg_get_small(cinfo, min_request + slop);
            if (hdr_ptr != NULL)
                break;
            slop /= 2;
            if (slop < MIN_SLOP) /* give up when it gets real small */
                out_of_memory(cinfo, 2); /* jpeg_get_small failed */
        }
        mem->total_space_allocated += min_request + slop;
        /* Success, initialize the new pool header and add to end of list */
        hdr_ptr->hdr.next = NULL;
        hdr_ptr->hdr.bytes_used = 0;
        hdr_ptr->hdr.bytes_left = sizeobject + slop;
        if (prev_hdr_ptr == NULL) /* first pool in class? */
            mem->small_list[pool_id] = hdr_ptr;
        else
            prev_hdr_ptr->hdr.next = hdr_ptr;
    }

    /* OK, allocate the object from the current pool */
    data_ptr = (char *) (hdr_ptr + 1); /* point to first data byte in pool */
    data_ptr += hdr_ptr->hdr.bytes_used; /* point to place for object */
    hdr_ptr->hdr.bytes_used += sizeobject;

```



```

hdr_ptr->hdr.bytes_left -= sizeofobject;

return (void *) data_ptr;
}

/*
 * Allocation of "large" objects.
 *
 * The external semantics of these are the same as "small" objects,
 * except that FAR pointers are used on 80x86. However the pool
 * management heuristics are quite different. We assume that each
 * request is large enough that it may as well be passed directly to
 * jpeg_get_large; the pool management just links everything together
 * so that we can free it all on demand.
 * Note: the major use of "large" objects is in JSAMPARRAY and JBLOCKARRAY
 * structures. The routines that create these structures (see below)
 * deliberately bunch rows together to ensure a large request size.
 */

METHODDEF(void *)
alloc_large (j_common_ptr cinfo, int pool_id, size_t sizeofobject)
/* Allocate a "large" object */
{
    my_mem_ptr mem = (my_mem_ptr) cinfo->mem;
    large_pool_ptr hdr_ptr;
    size_t odd_bytes;

    /* Check for unsatisfiable request (do now to ensure no overflow below) */
    if (sizeofobject > (size_t) (MAX_ALLOC_CHUNK-SIZEOF(large_pool_hdr)))
        out_of_memory(cinfo, 3); /* request exceeds malloc's ability */

    /* Round up the requested size to a multiple of SIZEOF(ALIGN_TYPE) */
    odd_bytes = sizeofobject % SIZEOF(ALIGN_TYPE);
    if (odd_bytes > 0)
        sizeofobject += SIZEOF(ALIGN_TYPE) - odd_bytes;

    /* Always make a new pool */
    if (pool_id < 0 || pool_id >= JPOOL_NUMPOOLS)
        ERREXIT1(cinfo, JERR_BAD_POOL_ID, pool_id); /* safety check */

    hdr_ptr = (large_pool_ptr) jpeg_get_large(cinfo, sizeofobject +
        SIZEOF(large_pool_hdr));
    if (hdr_ptr == NULL)
        out_of_memory(cinfo, 4); /* jpeg_get_large failed */
    mem->total_space_allocated += sizeofobject + SIZEOF(large_pool_hdr);

    /* Success, initialize the new pool header and add to list */
    hdr_ptr->hdr.next = mem->large_list[pool_id];
    /* We maintain space counts in each pool header for statistical purposes,
     * even though they are not needed for allocation.
     */
    hdr_ptr->hdr.bytes_used = sizeofobject;
    hdr_ptr->hdr.bytes_left = 0;
    mem->large_list[pool_id] = hdr_ptr;

    return (void *) (hdr_ptr + 1); /* point to first data byte in pool */
}

/*
 * Creation of 2-D sample arrays.
 * The pointers are in near heap, the samples themselves in FAR heap.
 *
 * To minimize allocation overhead and to allow I/O of large contiguous
 * blocks, we allocate the sample rows in groups of as many rows as possible
 * without exceeding MAX_ALLOC_CHUNK total bytes per allocation request.
 * NB: the virtual array control routines, later in this file, know about
 * this chunking of rows. The rowsperchunk value is left in the mem manager
 * object so that it can be saved away if this sarray is the workspace for
 * a virtual array.
 */

METHODDEF(JSAMPARRAY)
alloc_sarray (j_common_ptr cinfo, int pool_id,
              JDIMENSION samplesperrow, JDIMENSION numrows)
/* Allocate a 2-D sample array */
{
    my_mem_ptr mem = (my_mem_ptr) cinfo->mem;
    JSAMPARRAY result;

```

```

JSAMPROW workspace;
JDIMENSION rowsperchunk, currow, i;
long ltemp;

/* Calculate max # of rows allowed in one allocation chunk */
ltemp = (MAX_ALLOC_CHUNK-SIZEOF(large_pool_hdr)) /
    ((long) samplesperrow * SIZEOF(JSAMPLE));
if (ltemp <= 0)
    ERREXIT(cinfo, JERR_WIDTH_OVERFLOW);
if (ltemp < (long) numRows)
    rowsperchunk = (JDIMENSION) ltemp;
else
    rowsperchunk = numRows;
mem->last_rowsperchunk = rowsperchunk;

/* Get space for row pointers (small object) */
result = (JSAMPARRAY) alloc_small(cinfo, pool_id,
    (size_t) (numRows * SIZEOF(JSAMPROW)));

/* Get the rows themselves (large objects) */
currow = 0;
while (currow < numRows) {
    rowsperchunk = MIN(rowsperchunk, numRows - currow);
    workspace = (JSAMPROW) alloc_large(cinfo, pool_id,
        (size_t) ((size_t) rowsperchunk * (size_t) samplesperrow
            * SIZEOF(JSAMPLE)));
    for (i = rowsperchunk; i > 0; i--) {
        result[currow++] = workspace;
        workspace += samplesperrow;
    }
}

return result;
}

/*
 * Creation of 2-D coefficient-block arrays.
 * This is essentially the same as the code for sample arrays, above.
 */

METHODDEF(JBLOCKARRAY)
alloc_barray (j_common_ptr cinfo, int pool_id,
    JDIMENSION blocksperrow, JDIMENSION numRows)
/* Allocate a 2-D coefficient-block array */
{
    my_mem_ptr mem = (my_mem_ptr) cinfo->mem;
    JBLOCKARRAY result;
    JBLOCKROW workspace;
    JDIMENSION rowsperchunk, currow, i;
    long ltemp;

/* Calculate max # of rows allowed in one allocation chunk */
ltemp = (MAX_ALLOC_CHUNK-SIZEOF(large_pool_hdr)) /
    ((long) blocksperrow * SIZEOF(JBLOCK));
if (ltemp <= 0)
    ERREXIT(cinfo, JERR_WIDTH_OVERFLOW);
if (ltemp < (long) numRows)
    rowsperchunk = (JDIMENSION) ltemp;
else
    rowsperchunk = numRows;
mem->last_rowsperchunk = rowsperchunk;

/* Get space for row pointers (small object) */
result = (JBLOCKARRAY) alloc_small(cinfo, pool_id,
    (size_t) (numRows * SIZEOF(JBLOCKROW)));

/* Get the rows themselves (large objects) */
currow = 0;
while (currow < numRows) {
    rowsperchunk = MIN(rowsperchunk, numRows - currow);
    workspace = (JBLOCKROW) alloc_large(cinfo, pool_id,
        (size_t) ((size_t) rowsperchunk * (size_t) blocksperrow
            * SIZEOF(JBLOCK)));
    for (i = rowsperchunk; i > 0; i--) {
        result[currow++] = workspace;
        workspace += blocksperrow;
    }
}
}

```

```

    return result;
}

/*
 * About virtual array management:
 *
 * The above "normal" array routines are only used to allocate strip buffers
 * (as wide as the image, but just a few rows high). Full-image-sized buffers
 * are handled as "virtual" arrays. The array is still accessed a strip at a
 * time, but the memory manager must save the whole array for repeated
 * accesses. The intended implementation is that there is a strip buffer in
 * memory (as high as is possible given the desired memory limit), plus a
 * backing file that holds the rest of the array.
 *
 * The request_virt_array routines are told the total size of the image and
 * the maximum number of rows that will be accessed at once. The in-memory
 * buffer must be at least as large as the maxaccess value.
 *
 * The request routines create control blocks but not the in-memory buffers.
 * That is postponed until realize_virt_arrays is called. At that time the
 * total amount of space needed is known (approximately, anyway), so free
 * memory can be divided up fairly.
 *
 * The access_virt_array routines are responsible for making a specific strip
 * area accessible (after reading or writing the backing file, if necessary).
 * Note that the access routines are told whether the caller intends to modify
 * the accessed strip; during a read-only pass this saves having to rewrite
 * data to disk. The access routines are also responsible for pre-zeroing
 * any newly accessed rows, if pre-zeroing was requested.
 *
 * In current usage, the access requests are usually for nonoverlapping
 * strips; that is, successive access start_row numbers differ by exactly
 * num_rows = maxaccess. This means we can get good performance with simple
 * buffer dump/reload logic, by making the in-memory buffer be a multiple
 * of the access height; then there will never be accesses across bufferload
 * boundaries. The code will still work with overlapping access requests,
 * but it doesn't handle bufferload overlaps very efficiently.
 */

METHODDEF(jvirt_sarray_ptr)
request_virt_sarray (j_common_ptr cinfo, int pool_id, boolean pre_zero,
                    JDIMENSION samplesperrow, JDIMENSION numrows,
                    JDIMENSION maxaccess)
/* Request a virtual 2-D sample array */
{
    my_mem_ptr mem = (my_mem_ptr) cinfo->mem;
    jvirt_sarray_ptr result;

    /* Only IMAGE-lifetime virtual arrays are currently supported */
    if (pool_id != JPOOL_IMAGE)
        ERREXIT1(cinfo, JERR_BAD_POOL_ID, pool_id); /* safety check */

    /* get control block */
    result = (jvirt_sarray_ptr) alloc_small(cinfo, pool_id,
                                           sizeof(struct jvirt_sarray_control));

    result->mem_buffer = NULL; /* marks array not yet realized */
    result->rows_in_array = numrows;
    result->samplesperrow = samplesperrow;
    result->maxaccess = maxaccess;
    result->pre_zero = pre_zero;
    result->b_s_open = FALSE; /* no associated backing-store object */
    result->next = mem->virt_sarray_list; /* add to list of virtual arrays */
    mem->virt_sarray_list = result;

    return result;
}

METHODDEF(jvirt_barray_ptr)
request_virt_barray (j_common_ptr cinfo, int pool_id, boolean pre_zero,
                    JDIMENSION blocksperrow, JDIMENSION numrows,
                    JDIMENSION maxaccess)
/* Request a virtual 2-D coefficient-block array */
{
    my_mem_ptr mem = (my_mem_ptr) cinfo->mem;
    jvirt_barray_ptr result;

```

```

/* Only IMAGE-lifetime virtual arrays are currently supported */
if (pool_id != JPOOL_IMAGE)
    ERREXIT1(cinfo, JERR_BAD_POOL_ID, pool_id); /* safety check */

/* get control block */
result = (jvirt_barray_ptr) alloc_small(cinfo, pool_id,
    SIZEOF(struct jvirt_barray_control));

result->mem_buffer = NULL; /* marks array not yet realized */
result->rows_in_array = numrows;
result->blocksperrrow = blocksperrrow;
result->maxaccess = maxaccess;
result->pre_zero = pre_zero;
result->b_s_open = FALSE; /* no associated backing-store object */
result->next = mem->virt_barray_list; /* add to list of virtual arrays */
mem->virt_barray_list = result;

return result;
}

METHODDEF(void)
realize_virt_arrays (j_common_ptr cinfo)
/* Allocate the in-memory buffers for any unrealized virtual arrays */
{
    my_mem_ptr mem = (my_mem_ptr) cinfo->mem;
    long space_per_minheight, maximum_space, avail_mem;
    long minheights, max_minheights;
    jvirt_sarray_ptr sptr;
    jvirt_barray_ptr bptr;

    /* Compute the minimum space needed (maxaccess rows in each buffer)
     * and the maximum space needed (full image height in each buffer).
     * These may be of use to the system-dependent jpeg_mem_available routine.
     */
    space_per_minheight = 0;
    maximum_space = 0;
    for (sptr = mem->virt_sarray_list; sptr != NULL; sptr = sptr->next) {
        if (sptr->mem_buffer == NULL) { /* if not realized yet */
            space_per_minheight += (long) sptr->maxaccess *
                (long) sptr->samplesperrow * SIZEOF(JSAMPLE);
            maximum_space += (long) sptr->rows_in_array *
                (long) sptr->samplesperrow * SIZEOF(JSAMPLE);
        }
    }
    for (bptr = mem->virt_barray_list; bptr != NULL; bptr = bptr->next) {
        if (bptr->mem_buffer == NULL) { /* if not realized yet */
            space_per_minheight += (long) bptr->maxaccess *
                (long) bptr->blocksperrrow * SIZEOF(JBLOCK);
            maximum_space += (long) bptr->rows_in_array *
                (long) bptr->blocksperrrow * SIZEOF(JBLOCK);
        }
    }

    if (space_per_minheight <= 0)
        return; /* no unrealized arrays, no work */

    /* Determine amount of memory to actually use; this is system-dependent. */
    avail_mem = jpeg_mem_available(cinfo, space_per_minheight, maximum_space,
        mem->total_space_allocated);

    /* If the maximum space needed is available, make all the buffers full
     * height; otherwise parcel it out with the same number of minheights
     * in each buffer.
     */
    if (avail_mem >= maximum_space)
        max_minheights = 1000000000L;
    else {
        max_minheights = avail_mem / space_per_minheight;
        /* If there doesn't seem to be enough space, try to get the minimum
         * anyway. This allows a "stub" implementation of jpeg_mem_available().
         */
        if (max_minheights <= 0)
            max_minheights = 1;
    }

    /* Allocate the in-memory buffers and initialize backing store as needed. */
    for (sptr = mem->virt_sarray_list; sptr != NULL; sptr = sptr->next) {
        if (sptr->mem_buffer == NULL) { /* if not realized yet */

```

```

    minheights = ((long) sptr->rows_in_array - 1L) / sptr->maxaccess + 1L;
    if (minheights <= max_minheights) {
        /* This buffer fits in memory */
        sptr->rows_in_mem = sptr->rows_in_array;
    } else {
        /* It doesn't fit in memory, create backing store. */
        sptr->rows_in_mem = (JDIMENSION) (max_minheights * sptr->maxaccess);
        jpeg_open_backing_store(cinfo, & sptr->b_s_info,
            (long) sptr->rows_in_array *
            (long) sptr->samplesperrow *
            (long) SIZEOF(JSAMPLE));
        sptr->b_s_open = TRUE;
    }
    sptr->mem_buffer = alloc_sarray(cinfo, JPOOL_IMAGE,
        sptr->samplesperrow, sptr->rows_in_mem);
    sptr->rowsperchunk = mem->last_rowsperchunk;
    sptr->cur_start_row = 0;
    sptr->first_undef_row = 0;
    sptr->dirty = FALSE;
}

for (bp_ptr = mem->virt_barray_list; bp_ptr != NULL; bp_ptr = bp_ptr->next) {
    if (bp_ptr->mem_buffer == NULL) { /* if not realized yet */
        minheights = ((long) bp_ptr->rows_in_array - 1L) / bp_ptr->maxaccess + 1L;
        if (minheights <= max_minheights) {
            /* This buffer fits in memory */
            bp_ptr->rows_in_mem = bp_ptr->rows_in_array;
        } else {
            /* It doesn't fit in memory, create backing store. */
            bp_ptr->rows_in_mem = (JDIMENSION) (max_minheights * bp_ptr->maxaccess);
            jpeg_open_backing_store(cinfo, & bp_ptr->b_s_info,
                (long) bp_ptr->rows_in_array *
                (long) bp_ptr->blocksperrrow *
                (long) SIZEOF(JBLOCK));
            bp_ptr->b_s_open = TRUE;
        }
        bp_ptr->mem_buffer = alloc_barray(cinfo, JPOOL_IMAGE,
            bp_ptr->blocksperrrow, bp_ptr->rows_in_mem);
        bp_ptr->rowsperchunk = mem->last_rowsperchunk;
        bp_ptr->cur_start_row = 0;
        bp_ptr->first_undef_row = 0;
        bp_ptr->dirty = FALSE;
    }
}

LOCAL(void)
do_sarray_io (j_common_ptr cinfo, jvirt_sarray_ptr ptr, boolean writing)
/* Do backing store read or write of a virtual sample array */
{
    long bytesperrow, file_offset, byte_count, rows, thisrow, i;

    bytesperrow = (long) ptr->samplesperrow * SIZEOF(JSAMPLE);
    file_offset = ptr->cur_start_row * bytesperrow;
    /* Loop to read or write each allocation chunk in mem_buffer */
    for (i = 0; i < (long) ptr->rows_in_mem; i += ptr->rowsperchunk) {
        /* One chunk, but check for short chunk at end of buffer */
        rows = MIN((long) ptr->rowsperchunk, (long) ptr->rows_in_mem - i);
        /* Transfer no more than is currently defined */
        thisrow = (long) ptr->cur_start_row + i;
        rows = MIN(rows, (long) ptr->first_undef_row - thisrow);
        /* Transfer no more than fits in file */
        rows = MIN(rows, (long) ptr->rows_in_array - thisrow);
        if (rows <= 0) /* this chunk might be past end of file! */
            break;
        byte_count = rows * bytesperrow;
        if (writing)
            (*ptr->b_s_info.write_backing_store) (cinfo, & ptr->b_s_info,
                (void *) ptr->mem_buffer[i],
                file_offset, byte_count);
        else
            (*ptr->b_s_info.read_backing_store) (cinfo, & ptr->b_s_info,
                (void *) ptr->mem_buffer[i],
                file_offset, byte_count);
        file_offset += byte_count;
    }
}

```

```

LOCAL(void)
do_barray_io (j_common_ptr cinfo, jvirt_barray_ptr ptr, boolean writing)
/* Do backing store read or write of a virtual coefficient-block array */
{
    long bytesperrow, file_offset, byte_count, rows, thisrow, i;

    bytesperrow = (long) ptr->blocksperrow * SIZEOF(JBLOCK);
    file_offset = ptr->cur_start_row * bytesperrow;
    /* Loop to read or write each allocation chunk in mem_buffer */
    for (i = 0; i < (long) ptr->rows_in_mem; i += ptr->rowsperchunk) {
        /* One chunk, but check for short chunk at end of buffer */
        rows = MIN((long) ptr->rowsperchunk, (long) ptr->rows_in_mem - i);
        /* Transfer no more than is currently defined */
        thisrow = (long) ptr->cur_start_row + i;
        rows = MIN(rows, (long) ptr->first_undef_row - thisrow);
        /* Transfer no more than fits in file */
        rows = MIN(rows, (long) ptr->rows_in_array - thisrow);
        if (rows <= 0) /* this chunk might be past end of file! */
            break;
        byte_count = rows * bytesperrow;
        if (writing)
            (*ptr->b_s_info.write_backing_store) (cinfo, & ptr->b_s_info,
                                                  (void *) ptr->mem_buffer[i],
                                                  file_offset, byte_count);
        else
            (*ptr->b_s_info.read_backing_store) (cinfo, & ptr->b_s_info,
                                                  (void *) ptr->mem_buffer[i],
                                                  file_offset, byte_count);
        file_offset += byte_count;
    }
}

METHODDEF(JSAMPARRAY)
access_virt_sarray (j_common_ptr cinfo, jvirt_sarray_ptr ptr,
                    JDIMENSION start_row, JDIMENSION num_rows,
                    boolean writable)
/* Access the part of a virtual sample array starting at start_row */
/* and extending for num_rows rows. writable is true if */
/* caller intends to modify the accessed area. */
{
    JDIMENSION end_row = start_row + num_rows;
    JDIMENSION undef_row;

    /* debugging check */
    if (end_row > ptr->rows_in_array || num_rows > ptr->maxaccess ||
        ptr->mem_buffer == NULL)
        ERREXIT(cinfo, JERR_BAD_VIRTUAL_ACCESS);

    /* Make the desired part of the virtual array accessible */
    if (start_row < ptr->cur_start_row ||
        end_row > ptr->cur_start_row + ptr->rows_in_mem) {
        if (! ptr->b_s_open)
            ERREXIT(cinfo, JERR_VIRTUAL_BUG);
        /* Flush old buffer contents if necessary */
        if (ptr->dirty) {
            do_sarray_io(cinfo, ptr, TRUE);
            ptr->dirty = FALSE;
        }
        /* Decide what part of virtual array to access.
         * Algorithm: if target address > current window, assume forward scan,
         * load starting at target address. If target address < current window,
         * assume backward scan, load so that target area is top of window.
         * Note that when switching from forward write to forward read, will have
         * start_row = 0, so the limiting case applies and we load from 0 anyway.
         */
        if (start_row > ptr->cur_start_row) {
            ptr->cur_start_row = start_row;
        } else {
            /* use long arithmetic here to avoid overflow & unsigned problems */
            long ltemp;

            ltemp = (long) end_row - (long) ptr->rows_in_mem;
            if (ltemp < 0)
                ltemp = 0; /* don't fall off front end of file */
            ptr->cur_start_row = (JDIMENSION) ltemp;
        }
        /* Read in the selected part of the array.
         * During the initial write pass, we will do no actual read

```

```

    * because the selected part is all undefined.
    */
    do_sarray_io(cinfo, ptr, FALSE);
}
/* Ensure the accessed part of the array is defined; prezero if needed.
 * To improve locality of access, we only prezero the part of the array
 * that the caller is about to access, not the entire in-memory array.
 */
if (ptr->first_undef_row < end_row) {
    if (ptr->first_undef_row < start_row) {
        if (writable) /* writer skipped over a section of array */
            ERREXIT(cinfo, JERR_BAD_VIRTUAL_ACCESS);
        undef_row = start_row; /* but reader is allowed to read ahead */
    } else {
        undef_row = ptr->first_undef_row;
    }
    if (writable)
        ptr->first_undef_row = end_row;
    if (ptr->pre_zero) {
        size_t bytesperrow = (size_t) ptr->samplesperrow * sizeof(JSAMPLE);
        undef_row -= ptr->cur_start_row; /* make indexes relative to buffer */
        end_row -= ptr->cur_start_row;
        while (undef_row < end_row) {
            jzero_far((void *) ptr->mem_buffer[undef_row], bytesperrow);
            undef_row++;
        }
    } else {
        if (!writable) /* reader looking at undefined data */
            ERREXIT(cinfo, JERR_BAD_VIRTUAL_ACCESS);
    }
}

/* Flag the buffer dirty if caller will write in it */
if (writable)
    ptr->dirty = TRUE;
/* Return address of proper part of the buffer */
return ptr->mem_buffer + (start_row - ptr->cur_start_row);
}

METHODDEF(JBLOCKARRAY)
access_virt_barray(j_common_ptr cinfo, jvirt_barray_ptr ptr,
                  JDIMENSION start_row, JDIMENSION num_rows,
                  boolean writable)
/* Access the part of a virtual block array starting at start_row */
/* and extending for num_rows rows. writable is true if */
/* caller intends to modify the accessed area. */
{
    JDIMENSION end_row = start_row + num_rows;
    JDIMENSION undef_row;

    /* debugging check */
    if (end_row > ptr->rows_in_array || num_rows > ptr->maxaccess ||
        ptr->mem_buffer == NULL)
        ERREXIT(cinfo, JERR_BAD_VIRTUAL_ACCESS);

    /* Make the desired part of the virtual array accessible */
    if (start_row < ptr->cur_start_row ||
        end_row > ptr->cur_start_row + ptr->rows_in_mem) {
        if (!ptr->b_s_open)
            ERREXIT(cinfo, JERR_VIRTUAL_BUG);
        /* Flush old buffer contents if necessary */
        if (ptr->dirty) {
            do_barray_io(cinfo, ptr, TRUE);
            ptr->dirty = FALSE;
        }
    }
    /* Decide what part of virtual array to access.
     * Algorithm: if target address > current window, assume forward scan,
     * load starting at target address. If target address < current window,
     * assume backward scan, load so that target area is top of window.
     * Note that when switching from forward write to forward read, will have
     * start_row = 0, so the limiting case applies and we load from 0 anyway.
     */
    if (start_row > ptr->cur_start_row) {
        ptr->cur_start_row = start_row;
    } else {
        /* use long arithmetic here to avoid overflow & unsigned problems */
        long ltemp;

        ltemp = (long) end_row - (long) ptr->rows_in_mem;
        if (ltemp < 0)

```

```

ltemp = 0; /* don't roll off front end of file */
ptr->cur_start_row = (JMEMENSION) ltemp;
}
/* Read in the selected part of the array.
 * During the initial write pass, we will do no actual read
 * because the selected part is all undefined.
 */
do_barray_io(cinfo, ptr, FALSE);
}
/* Ensure the accessed part of the array is defined; prezero if needed.
 * To improve locality of access, we only prezero the part of the array
 * that the caller is about to access, not the entire in-memory array.
 */
if (ptr->first_undef_row < end_row) {
    if (ptr->first_undef_row < start_row) {
        if (writable) /* writer skipped over a section of array */
            ERREXIT(cinfo, JERR_BAD_VIRTUAL_ACCESS);
        undef_row = start_row; /* but reader is allowed to read ahead */
    } else {
        undef_row = ptr->first_undef_row;
    }
    if (writable)
        ptr->first_undef_row = end_row;
    if (ptr->pre_zero) {
        size_t bytesperrow = (size_t) ptr->blocksperrrow * SIZEOF(JBLOCK);
        undef_row -= ptr->cur_start_row; /* make indexes relative to buffer */
        end_row -= ptr->cur_start_row;
        while (undef_row < end_row) {
            jzero_far((void *) ptr->mem_buffer[undef_row], bytesperrow);
            undef_row++;
        }
    } else {
        if (!writable) /* reader looking at undefined data */
            ERREXIT(cinfo, JERR_BAD_VIRTUAL_ACCESS);
    }
}
/* Flag the buffer dirty if caller will write in it */
if (writable)
    ptr->dirty = TRUE;
/* Return address of proper part of the buffer */
return ptr->mem_buffer + (start_row - ptr->cur_start_row);
}

/*
 * Release all objects belonging to a specified pool.
 */
METHODDEF(void)
free_pool (j_common_ptr cinfo, int pool_id)
{
    my_mem_ptr mem = (my_mem_ptr) cinfo->mem;
    small_pool_ptr shdr_ptr;
    large_pool_ptr lhdr_ptr;
    size_t space_freed;

    if (pool_id < 0 || pool_id >= JPOOL_NUMPOOLS)
        ERREXIT1(cinfo, JERR_BAD_POOL_ID, pool_id); /* safety check */

#ifdef MEM_STATS
    if (cinfo->err->trace_level > 1)
        print_mem_stats(cinfo, pool_id); /* print pool's memory usage statistics */
#endif

    /* If freeing IMAGE pool, close any virtual arrays first */
    if (pool_id == JPOOL_IMAGE) {
        jvirt_sarray_ptr sptr;
        jvirt_barray_ptr bptr;

        for (sptr = mem->virt_sarray_list; sptr != NULL; sptr = sptr->next) {
            if (sptr->b_s_open) { /* there may be no backing store */
                sptr->b_s_open = FALSE; /* prevent recursive close if error */
                (*sptr->b_s_info.close_backing_store) (cinfo, & sptr->b_s_info);
            }
        }
        mem->virt_sarray_list = NULL;
        for (bptr = mem->virt_barray_list; bptr != NULL; bptr = bptr->next) {
            if (bptr->b_s_open) { /* there may be no backing store */
                bptr->b_s_open = FALSE; /* prevent recursive close if error */
                (*bptr->b_s_info.close_backing_store) (cinfo, & bptr->b_s_info);
            }
        }
    }
}

```



```

    }
    mem->virt_barray_list = NULL;
}

/* Release large objects */
lhdr_ptr = mem->large_list[pool_id];
mem->large_list[pool_id] = NULL;

while (lhdr_ptr != NULL) {
    large_pool_ptr next_lhdr_ptr = lhdr_ptr->hdr.next;
    space_freed = lhdr_ptr->hdr.bytes_used +
        lhdr_ptr->hdr.bytes_left +
        SIZEOF(large_pool_hdr);
    jpeg_free_large(cinfo, (void *) lhdr_ptr, space_freed);
    mem->total_space_allocated -= space_freed;
    lhdr_ptr = next_lhdr_ptr;
}

/* Release small objects */
shdr_ptr = mem->small_list[pool_id];
mem->small_list[pool_id] = NULL;

while (shdr_ptr != NULL) {
    small_pool_ptr next_shdr_ptr = shdr_ptr->hdr.next;
    space_freed = shdr_ptr->hdr.bytes_used +
        shdr_ptr->hdr.bytes_left +
        SIZEOF(small_pool_hdr);
    jpeg_free_small(cinfo, (void *) shdr_ptr, space_freed);
    mem->total_space_allocated -= space_freed;
    shdr_ptr = next_shdr_ptr;
}

Close up shop entirely.
Note that this cannot be called unless cinfo->mem is non-NULL.
*/
METHODDEF(void)
self_destruct (j_common_ptr cinfo)
{
    int pool;

    /* Close all backing store, release all memory.
     * Releasing pools in reverse order might help avoid fragmentation
     * with some (brain-damaged) malloc libraries.
     */
    for (pool = JPOOL_NUMPOOLS-1; pool >= JPOOL_PERMANENT; pool--) {
        free_pool(cinfo, pool);
    }

    /* Release the memory manager control block too. */
    jpeg_free_small(cinfo, (void *) cinfo->mem, SIZEOF(my_memory_mgr));
    cinfo->mem = NULL; /* ensures I will be called only once */

    jpeg_mem_term(cinfo); /* system-dependent cleanup */
}

/*
 * Memory manager initialization.
 * When this is called, only the error manager pointer is valid in cinfo!
 */
GLOBAL(void)
jinit_memory_mgr (j_common_ptr cinfo)
{
    my_mem_ptr mem;
    long max_to_use;
    int pool;
    size_t test_mac;

    cinfo->mem = NULL; /* for safety if init fails */

    /* Check for configuration errors.
     * SIZEOF(ALIGN_TYPE) should be a power of 2; otherwise, it probably
     * doesn't reflect any real hardware alignment requirement.
     * The test is a little tricky: for X>0, X and X-1 have no one-bits

```

```

* in common if and only if is a power of 2, ie has only one bit.
* Some compilers may give "unreachable code" warning here; ignore it.
*/
if ((sizeof(ALIGN_TYPE) & (sizeof(ALIGN_TYPE)-1)) != 0)
    ERREXIT(cinfo, JERR_BAD_ALIGN_TYPE);
/* MAX_ALLOC_CHUNK must be representable as type size_t, and must be
* a multiple of sizeof(ALIGN_TYPE).
* Again, an "unreachable code" warning may be ignored here.
* But a "constant too large" warning means you need to fix MAX_ALLOC_CHUNK.
*/
test_mac = (size_t) MAX_ALLOC_CHUNK;
if ((long) test_mac != MAX_ALLOC_CHUNK ||
    (MAX_ALLOC_CHUNK % sizeof(ALIGN_TYPE)) != 0)
    ERREXIT(cinfo, JERR_BAD_ALLOC_CHUNK);

max_to_use = jpeg_mem_init(cinfo); /* system-dependent initialization */

/* Attempt to allocate memory manager's control block */
mem = (my_mem_ptr) jpeg_get_small(cinfo, sizeof(my_memory_mgr));

if (mem == NULL) {
    jpeg_mem_term(cinfo); /* system-dependent cleanup */
    ERREXIT1(cinfo, JERR_OUT_OF_MEMORY, 0);
}

/* OK, fill in the method pointers */
mem->pub.alloc_small = alloc_small;
mem->pub.alloc_large = alloc_large;
mem->pub.alloc_sarray = alloc_sarray;
mem->pub.alloc_barray = alloc_barray;
mem->pub.request_virt_sarray = request_virt_sarray;
mem->pub.request_virt_barray = request_virt_barray;
mem->pub.realize_virt_arrays = realize_virt_arrays;
mem->pub.access_virt_sarray = access_virt_sarray;
mem->pub.access_virt_barray = access_virt_barray;
mem->pub.free_pool = free_pool;
mem->pub.self_destruct = self_destruct;

/* Make MAX_ALLOC_CHUNK accessible to other modules */
mem->pub.max_alloc_chunk = MAX_ALLOC_CHUNK;

/* Initialize working state */
mem->pub.max_memory_to_use = max_to_use;

for (pool = JPOOL_NUMPOOLS-1; pool >= JPOOL_PERMANENT; pool--) {
    mem->small_list[pool] = NULL;
    mem->large_list[pool] = NULL;
}
mem->virt_sarray_list = NULL;
mem->virt_barray_list = NULL;

mem->total_space_allocated = sizeof(my_memory_mgr);

/* Declare ourselves open for business */
cinfo->mem = & mem->pub;

/* Check for an environment variable JPEGMEM; if found, override the
* default max_memory setting from jpeg_mem_init. Note that the
* surrounding application may again override this value.
* If your system doesn't support getenv(), define NO_GETENV to disable
* this feature.
*/
#ifdef NO_GETENV
{ char * memenv;

    if ((memenv = getenv("JPEGMEM")) != NULL) {
        char ch = 'x';

        if (sscanf(memenv, "%ld%c", &max_to_use, &ch) > 0) {
            if (ch == 'm' || ch == 'M')
                max_to_use *= 1000L;
            mem->pub.max_memory_to_use = max_to_use * 1000L;
        }
    }
}
#endif
}

```

```

/*
 * jmemnobs.c
 *
 * Copyright (C) 1992-1996, Thomas G. Lane.
 * This file is part of the Independent JPEG Group's software.
 * For conditions of distribution and use, see the accompanying README file.
 *
 * This file provides a really simple implementation of the system-
 * dependent portion of the JPEG memory manager.  This implementation
 * assumes that no backing-store files are needed: all required space
 * can be obtained from malloc().
 * This is very portable in the sense that it'll compile on almost anything,
 * but you'd better have lots of main memory (or virtual memory) if you want
 * to process big images.
 * Note that the max_memory_to_use option is ignored by this implementation.
 */

```

```

#define JPEG_INTERNALS
#include "jinclude.h"
#include "jpeglib.h"
#include "jmemsys.h"          /* import the system-dependent declarations */

#ifdef HAVE_STDLIB_H           /* <stdlib.h> should declare malloc(),free() */
extern void * malloc JPP((size_t size));
extern void free JPP((void *ptr));
#endif

```

```

/*
 * Memory allocation and freeing are controlled by the regular library
 * routines malloc() and free().
 */

```

```

GLOBAL(void *)
jpeg_get_small (j_common_ptr cinfo, size_t sizeofobject)
{
    return (void *) malloc(sizeofobject);
}

```

```

GLOBAL(void)
jpeg_free_small (j_common_ptr cinfo, void * object, size_t sizeofobject)
{
    free(object);
}

```

```

/*
 * "Large" objects are treated the same as "small" ones.
 * NB: although we include FAR keywords in the routine declarations,
 * this file won't actually work in 80x86 small/medium model; at least,
 * you probably won't be able to process useful-size images in only 64KB.
 */

```

```

GLOBAL(void *)
jpeg_get_large (j_common_ptr cinfo, size_t sizeofobject)
{
    return (void *) malloc(sizeofobject);
}

```

```

GLOBAL(void)
jpeg_free_large (j_common_ptr cinfo, void * object, size_t sizeofobject)
{
    free(object);
}

```

```

/*
 * This routine computes the total memory space available for allocation.
 * Here we always say, "we got all you want bud!"
 */

```

```

GLOBAL(long)
jpeg_mem_available (j_common_ptr cinfo, long min_bytes_needed,
                   long max_bytes_needed, long already_allocated)
{
    return max_bytes_needed;
}

```

```

/*

```



```

/*
 * jquant1.c
 *
 * Copyright (C) 1991-1996, Thomas G. Lane.
 * This file is part of the Independent JPEG Group's software.
 * For conditions of distribution and use, see the accompanying README file.
 *
 * This file contains 1-pass color quantization (color mapping) routines.
 * These routines provide mapping to a fixed color map using equally spaced
 * color values. Optional Floyd-Steinberg or ordered dithering is available.
 */

#define JPEG_INTERNALS
#include "jinclude.h"
#include "jpeglib.h"

#ifdef QUANT_1PASS_SUPPORTED

/*
 * The main purpose of 1-pass quantization is to provide a fast, if not very
 * high quality, colormapped output capability. A 2-pass quantizer usually
 * gives better visual quality; however, for quantized grayscale output this
 * quantizer is perfectly adequate. Dithering is highly recommended with this
 * quantizer, though you can turn it off if you really want to.
 *
 * In 1-pass quantization the colormap must be chosen in advance of seeing the
 * image. We use a map consisting of all combinations of Ncolors[i] color
 * values for the i'th component. The Ncolors[] values are chosen so that
 * their product, the total number of colors, is no more than that requested.
 * (In most cases, the product will be somewhat less.)
 *
 * Since the colormap is orthogonal, the representative value for each color
 * component can be determined without considering the other components;
 * then these indexes can be combined into a colormap index by a standard
 * N-dimensional-array-subscript calculation. Most of the arithmetic involved
 * can be precalculated and stored in the lookup table colorindex[].
 * colorindex[i][j] maps pixel value j in component i to the nearest
 * representative value (grid plane) for that component; this index is
 * multiplied by the array stride for component i, so that the
 * index of the colormap entry closest to a given pixel value is just
 * sum( colorindex[component-number][pixel-component-value] )
 * Aside from being fast, this scheme allows for variable spacing between
 * representative values with no additional lookup cost.
 *
 * If gamma correction has been applied in color conversion, it might be wise
 * to adjust the color grid spacing so that the representative colors are
 * equidistant in linear space. At this writing, gamma correction is not
 * implemented by jdcolor, so nothing is done here.
 */

/* Declarations for ordered dithering.
 *
 * We use a standard 16x16 ordered dither array. The basic concept of ordered
 * dithering is described in many references, for instance Dale Schumacher's
 * chapter II.2 of Graphics Gems II (James Arvo, ed. Academic Press, 1991).
 * In place of Schumacher's comparisons against a "threshold" value, we add a
 * "dither" value to the input pixel and then round the result to the nearest
 * output value. The dither value is equivalent to (0.5 - threshold) times
 * the distance between output values. For ordered dithering, we assume that
 * the output colors are equally spaced; if not, results will probably be
 * worse, since the dither may be too much or too little at a given point.
 *
 * The normal calculation would be to form pixel value + dither, range-limit
 * this to 0..MAXJSAMPLE, and then index into the colorindex table as usual.
 * We can skip the separate range-limiting step by extending the colorindex
 * table in both directions.
 */

#define ODITHER_SIZE 16 /* dimension of dither matrix */
/* NB: if ODITHER_SIZE is not a power of 2, ODITHER_MASK uses will break */
#define ODITHER_CELLS (ODITHER_SIZE*ODITHER_SIZE) /* # cells in matrix */
#define ODITHER_MASK (ODITHER_SIZE-1) /* mask for wrapping around counters */

typedef int ODITHER_MATRIX[ODITHER_SIZE][ODITHER_SIZE];
typedef int (*ODITHER_MATRIX_PTR)[ODITHER_SIZE];

static const UINT8 base_dither_matrix[ODITHER_SIZE][ODITHER_SIZE] = {
  /* Bayer's order-4 dither array. Generated by the code given in

```

```

* Stephen Hawley's article "Ordered Dithering" in Graphics Gems
* The values in this array must range from 0 to ODITHER_CELLS-
*/
{ 0,192, 48,240, 12,204, 60,252, 3,195, 51,243, 15,207, 63,255 },
{ 128, 64,176,112,140, 76,188,124,131, 67,179,115,143, 79,191,127 },
{ 32,224, 16,208, 44,236, 28,220, 35,227, 19,211, 47,239, 31,223 },
{ 160, 96,144, 80,172,108,156, 92,163, 99,147, 83,175,111,159, 95 },
{ 8,200, 56,248, 4,196, 52,244, 11,203, 59,251, 7,199, 55,247 },
{ 136, 72,184,120,132, 68,180,116,139, 75,187,123,135, 71,183,119 },
{ 40,232, 24,216, 36,228, 20,212, 43,235, 27,219, 39,231, 23,215 },
{ 168,104,152, 88,164,100,148, 84,171,107,155, 91,167,103,151, 87 },
{ 2,194, 50,242, 14,206, 62,254, 1,193, 49,241, 13,205, 61,253 },
{ 130, 66,178,114,142, 78,190,126,129, 65,177,113,141, 77,189,125 },
{ 34,226, 18,210, 46,238, 30,222, 33,225, 17,209, 45,237, 29,221 },
{ 162, 98,146, 82,174,110,158, 94,161, 97,145, 81,173,109,157, 93 },
{ 10,202, 58,250, 6,198, 54,246, 9,201, 57,249, 5,197, 53,245 },
{ 138, 74,186,122,134, 70,182,118,137, 73,185,121,133, 69,181,117 },
{ 42,234, 26,218, 38,230, 22,214, 41,233, 25,217, 37,229, 21,213 },
{ 170,106,154, 90,166,102,150, 86,169,105,153, 89,165,101,149, 85 }
};

/* Declarations for Floyd-Steinberg dithering.
*
* Errors are accumulated into the array fserrors[], at a resolution of
* 1/16th of a pixel count. The error at a given pixel is propagated
* to its not-yet-processed neighbors using the standard F-S fractions,
* ... (here) 7/16
*           3/16 5/16 1/16
* We work left-to-right on even rows, right-to-left on odd rows.
*
* We can get away with a single array (holding one row's worth of errors)
* by using it to store the current row's errors at pixel columns not yet
* processed, but the next row's errors at columns already processed. We
* need only a few extra variables to hold the errors immediately around the
* current column. (If we are lucky, those variables are in registers, but
* even if not, they're probably cheaper to access than array elements are.)
*
* The fserrors[] array is indexed [component#][position].
* We provide (#columns + 2) entries per component; the extra entry at each
* end saves us from special-casing the first and last pixels.
*
* Note: on a wide image, we might not have enough room in a PC's near data
* segment to hold the error array; so it is allocated with alloc_large.
*/

#if BITS_IN_JSAMPLE == 8
typedef INT16 FSERROR; /* 16 bits should be enough */
typedef int LOCFSError; /* use 'int' for calculation temps */
#else
typedef INT32 FSERROR; /* may need more than 16 bits */
typedef INT32 LOCFSError; /* be sure calculation temps are big enough */
#endif

typedef FSERROR *FSERRPTR; /* pointer to error array (in FAR storage!) */

/* Private subobject */

#define MAX_Q_COMPS 4 /* max components I can handle */

typedef struct {
    struct jpeg_color_quantizer pub; /* public fields */

    /* Initially allocated colormap is saved here */
    JSAMPARRAY sv_colormap; /* The color map as a 2-D pixel array */
    int sv_actual; /* number of entries in use */

    JSAMPARRAY colorindex; /* Precomputed mapping for speed */
    /* colorindex[i][j] = index of color closest to pixel value j in component i,
     * premultiplied as described above. Since colormap indexes must fit into
     * JSAMPLEs, the entries of this array will too.
     */
    boolean is_padded; /* is the colorindex padded for odither? */

    int Ncolors[MAX_Q_COMPS]; /* # of values allocated to each component */

    /* Variables for ordered dithering */
    int row_index; /* cur row's vertical index in dither matrix */
    ODITHER_MATRIX_PTR odither[MAX_Q_COMPS]; /* one dither array per component */

```

```

/* Variables for Floyd-Steinberg dithering */
FSERRPTR fserrors[MAX_Q_COLORS]; /* accumulated errors */
boolean on_odd_row; /* flag to remember which row we are on */
) my_cquantizer;

typedef my_cquantizer * my_cquantize_ptr;

/*
 * Policy-making subroutines for create_colormap and create_colorindex.
 * These routines determine the colormap to be used. The rest of the module
 * only assumes that the colormap is orthogonal.
 *
 * * select_ncolors decides how to divvy up the available colors
 * among the components.
 * * output_value defines the set of representative values for a component.
 * * largest_input_value defines the mapping from input values to
 * representative values for a component.
 * Note that the latter two routines may impose different policies for
 * different components, though this is not currently done.
 */

LOCAL(int)
select_ncolors (j_decompress_ptr cinfo, int Ncolors[])
/* Determine allocation of desired colors to components, */
/* and fill in Ncolors[] array to indicate choice. */
/* Return value is total number of colors (product of Ncolors[] values). */
{
    int nc = cinfo->out_color_components; /* number of color components */
    int max_colors = cinfo->desired_number_of_colors;
    int total_colors, iroot, i, j;
    boolean changed;
    long temp;
    static const int RGB_order[3] = { RGB_GREEN, RGB_RED, RGB_BLUE };

    /* We can allocate at least the nc'th root of max_colors per component. */
    /* Compute floor(nc'th root of max_colors). */
    iroot = 1;
    do {
        iroot++;
        temp = iroot; /* set temp = iroot ** nc */
        for (i = 1; i < nc; i++)
            temp *= iroot;
    } while (temp <= (long) max_colors); /* repeat till iroot exceeds root */
    iroot--; /* now iroot = floor(root) */

    /* Must have at least 2 color values per component */
    if (iroot < 2)
        ERREXIT1(cinfo, JERR_QUANT_FEW_COLORS, (int) temp);

    /* Initialize to iroot color values for each component */
    total_colors = 1;
    for (i = 0; i < nc; i++) {
        Ncolors[i] = iroot;
        total_colors *= iroot;
    }

    /* We may be able to increment the count for one or more components without
     * exceeding max_colors, though we know not all can be incremented.
     * Sometimes, the first component can be incremented more than once!
     * (Example: for 16 colors, we start at 2*2*2, go to 3*2*2, then 4*2*2.)
     * In RGB colorspace, try to increment G first, then R, then B.
     */
    do {
        changed = FALSE;
        for (i = 0; i < nc; i++) {
            j = (cinfo->out_color_space == JCS_RGB ? RGB_order[i] : i);
            /* calculate new total_colors if Ncolors[j] is incremented */
            temp = total_colors / Ncolors[j];
            temp *= Ncolors[j] + 1; /* done in long arith to avoid oflo */
            if (temp > (long) max_colors)
                break; /* won't fit, done with this pass */
            Ncolors[j]++; /* OK, apply the increment */
            total_colors = (int) temp;
            changed = TRUE;
        }
    } while (changed);

    return total_colors;
}

```

```

)

LOCAL(int)
output_value (j_decompress_ptr cinfo, int ci, int j, int maxj)
/* Return j'th output value, where j will range from 0 to maxj */
/* The output values must fall in 0..MAXJSAMPLE in increasing order */
{
    /* We always provide values 0 and MAXJSAMPLE for each component;
     * any additional values are equally spaced between these limits.
     * (Forcing the upper and lower values to the limits ensures that
     * dithering can't produce a color outside the selected gamut.)
     */
    return (int) (((INT32) j * MAXJSAMPLE + maxj/2) / maxj);
}

LOCAL(int)
largest_input_value (j_decompress_ptr cinfo, int ci, int j, int maxj)
/* Return largest input value that should map to j'th output value */
/* Must have largest(j=0) >= 0, and largest(j=maxj) >= MAXJSAMPLE */
{
    /* Breakpoints are halfway between values returned by output_value */
    return (int) (((INT32) (2*j + 1) * MAXJSAMPLE + maxj) / (2*maxj));
}

/*
 * Create the colormap.
 */
LOCAL(void)
create_colormap (j_decompress_ptr cinfo)
{
    my_cquantize_ptr cquantize = (my_cquantize_ptr) cinfo->cquantize;
    JSAMPARRAY colormap; /* Created colormap */
    int total_colors; /* Number of distinct output colors */
    int i,j,k, nci, blksize, blkdist, ptr, val;

    /* Select number of colors for each component */
    total_colors = select_ncolors(cinfo, cquantize->Ncolors);

    /* Report selected color counts */
    if (cinfo->out_color_components == 3)
        TRACE4(cinfo, 1, JTRC_QUANT_3_NCOLORS,
              total_colors, cquantize->Ncolors[0],
              cquantize->Ncolors[1], cquantize->Ncolors[2]);
    else
        TRACE1(cinfo, 1, JTRC_QUANT_NCOLORS, total_colors);

    /* Allocate and fill in the colormap. */
    /* The colors are ordered in the map in standard row-major order, */
    /* i.e. rightmost (highest-indexed) color changes most rapidly. */

    colormap = (*cinfo->mem->alloc_sarray)
        ((j_common_ptr) cinfo, JPOOL_IMAGE,
         (JDIMENSION) total_colors, (JDIMENSION) cinfo->out_color_components);

    /* blksize is number of adjacent repeated entries for a component */
    /* blkdist is distance between groups of identical entries for a component */
    blkdist = total_colors;

    for (i = 0; i < cinfo->out_color_components; i++) {
        /* fill in colormap entries for i'th color component */
        nci = cquantize->Ncolors[i]; /* # of distinct values for this color */
        blksize = blkdist / nci;
        for (j = 0; j < nci; j++) {
            /* Compute j'th output value (out of nci) for component */
            val = output_value(cinfo, i, j, nci-1);
            /* Fill in all colormap entries that have this value of this component */
            for (ptr = j * blksize; ptr < total_colors; ptr += blkdist) {
                /* fill in blksize entries beginning at ptr */
                for (k = 0; k < blksize; k++)
                    colormap[i][ptr+k] = (JSAMPLE) val;
            }
            blkdist = blksize; /* blksize of this color is blkdist of next */
        }
    }

    /* Save the colormap in private storage,

```



```

    * where it will survive color quantization mode changes.
    */
    cquantize->sv_colormap = colormap;
    cquantize->sv_actual = total_colors;
}

/*
 * Create the color index table.
 */

LOCAL(void)
create_colorindex (j_decompress_ptr cinfo)
{
    my_cquantize_ptr cquantize = (my_cquantize_ptr) cinfo->cquantize;
    JSAMPROW indexptr;
    int i,j,k, nci, blksize, val, pad;

    /* For ordered dither, we pad the color index tables by MAXJSAMPLE in
     * each direction (input index values can be -MAXJSAMPLE .. 2*MAXJSAMPLE).
     * This is not necessary in the other dithering modes. However, we
     * flag whether it was done in case user changes dithering mode.
     */
    if (cinfo->dither_mode == JDITHER_ORDERED) {
        pad = MAXJSAMPLE*2;
        cquantize->is_padded = TRUE;
    } else {
        pad = 0;
        cquantize->is_padded = FALSE;
    }

    cquantize->colorindex = (*cinfo->mem->alloc_sarray)
        ((j_common_ptr) cinfo, JPOOL_IMAGE,
         (JDIMENSION) (MAXJSAMPLE+1 + pad),
         (JDIMENSION) cinfo->out_color_components);

    /* blksize is number of adjacent repeated entries for a component */
    blksize = cquantize->sv_actual;

    for (i = 0; i < cinfo->out_color_components; i++) {
        /* fill in colorindex entries for i'th color component */
        nci = cquantize->Ncolors[i]; /* # of distinct values for this color */
        blksize = blksize / nci;

        /* adjust colorindex pointers to provide padding at negative indexes. */
        if (pad)
            cquantize->colorindex[i] += MAXJSAMPLE;

        /* in loop, val = index of current output value, */
        /* and k = largest j that maps to current val */
        indexptr = cquantize->colorindex[i];
        val = 0;
        k = largest_input_value(cinfo, i, 0, nci-1);
        for (j = 0; j <= MAXJSAMPLE; j++) {
            while (j > k) /* advance val if past boundary */
                k = largest_input_value(cinfo, i, ++val, nci-1);
            /* premultiply so that no multiplication needed in main processing */
            indexptr[j] = (JSAMPLE) (val * blksize);
        }
        /* Pad at both ends if necessary */
        if (pad)
            for (j = 1; j <= MAXJSAMPLE; j++) {
                indexptr[-j] = indexptr[0];
                indexptr[MAXJSAMPLE+j] = indexptr[MAXJSAMPLE];
            }
    }
}

/*
 * Create an ordered-dither array for a component having ncolors
 * distinct output values.
 */

LOCAL(ODITHER_MATRIX_PTR)
make_odither_array (j_decompress_ptr cinfo, int ncolors)
{
    ODITHER_MATRIX_PTR odither;
    int j,k;
    INT32 num,den;

```

```

odither = (ODITHER_MATRIX_PTR)
(*cinfo->mem->alloc_small((j_common_ptr) cinfo, JPOOL_IMAGE,
    sizeof(ODITHER_MATRIX)));
/* The inter-value distance for this color is MAXJSAMPLE/(ncolors-1).
 * Hence the dither value for the matrix cell with fill order f
 * (f=0..N-1) should be (N-1-2*f)/(2*N) * MAXJSAMPLE/(ncolors-1).
 * On 16-bit-int machine, be careful to avoid overflow.
 */
den = 2 * ODITHER_CELLS * ((INT32) (ncolors - 1));
for (j = 0; j < ODITHER_SIZE; j++) {
    for (k = 0; k < ODITHER_SIZE; k++) {
        num = ((INT32) (ODITHER_CELLS-1 - 2*((int)base_dither_matrix[j][k])))
            * MAXJSAMPLE;
        /* Ensure round towards zero despite C's lack of consistency
         * about rounding negative values in integer division...
         */
        odither[j][k] = (int) (num<0 ? -((-num)/den) : num/den);
    }
}
return odither;
}

```

```

/*
 * Create the ordered-dither tables.
 * Components having the same number of representative colors may
 * share a dither table.
 */

```

```

LOCAL(void)
create_odither_tables (j_decompress_ptr cinfo)
{
    my_cquantize_ptr cquantize = (my_cquantize_ptr) cinfo->cquantize;
    ODITHER_MATRIX_PTR odither;
    int i, j, nci;

    for (i = 0; i < cinfo->out_color_components; i++) {
        nci = cquantize->Ncolors[i]; /* # of distinct values for this color */
        odither = NULL; /* search for matching prior component */
        for (j = 0; j < i; j++) {
            if (nci == cquantize->Ncolors[j]) {
                odither = cquantize->odither[j];
                break;
            }
        }
        if (odither == NULL) /* need a new table? */
            odither = make_odither_array(cinfo, nci);
        cquantize->odither[i] = odither;
    }
}

/*
 * Map some rows of pixels to the output colormapped representation.
 */

```

```

METHODDEF(void)
color_quantize (j_decompress_ptr cinfo, JSAMPARRAY input_buf,
    JSAMPARRAY output_buf, int num_rows)
/* General case, no dithering */
{
    my_cquantize_ptr cquantize = (my_cquantize_ptr) cinfo->cquantize;
    JSAMPARRAY colorindex = cquantize->colorindex;
    register int pixcode, ci;
    register JSAMPROW ptrin, ptrout;
    int row;
    JDIMENSION col;
    JDIMENSION width = cinfo->output_width;
    register int nc = cinfo->out_color_components;

    for (row = 0; row < num_rows; row++) {
        ptrin = input_buf[row];
        ptrout = output_buf[row];
        for (col = width; col > 0; col--) {
            pixcode = 0;
            for (ci = 0; ci < nc; ci++) {
                pixcode += GETJSAMPLE(colorindex[ci][GETJSAMPLE(*ptrin++)]);
            }
            *ptrout++ = (JSAMPLE) pixcode;
        }
    }
}

```

```

METHODDEF(void)
color_quantize3 (j_decompress_ptr cinfo, JSAMPARRAY input_buf,
                 JSAMPARRAY output_buf, int num_rows)
/* Fast path for out_color_components==3, no dithering */
{
    my_cquantize_ptr cquantize = (my_cquantize_ptr) cinfo->cquantize;
    register int pixcode;
    register JSAMPROW ptrin, ptrout;
    JSAMPROW colorindex0 = cquantize->colorindex[0];
    JSAMPROW colorindex1 = cquantize->colorindex[1];
    JSAMPROW colorindex2 = cquantize->colorindex[2];
    int row;
    JDIMENSION col;
    JDIMENSION width = cinfo->output_width;

    for (row = 0; row < num_rows; row++) {
        ptrin = input_buf[row];
        ptrout = output_buf[row];
        for (col = width; col > 0; col--) {
            pixcode = GETJSAMPLE(colorindex0[GETJSAMPLE(*ptrin++)]);
            pixcode += GETJSAMPLE(colorindex1[GETJSAMPLE(*ptrin++)]);
            pixcode += GETJSAMPLE(colorindex2[GETJSAMPLE(*ptrin++)]);
            *ptrout++ = (JSAMPLE) pixcode;
        }
    }
}

METHODDEF(void)
color_quantize_ord_dither (j_decompress_ptr cinfo, JSAMPARRAY input_buf,
                          JSAMPARRAY output_buf, int num_rows)
/* General case, with ordered dithering */
{
    my_cquantize_ptr cquantize = (my_cquantize_ptr) cinfo->cquantize;
    register JSAMPROW input_ptr;
    register JSAMPROW output_ptr;
    JSAMPROW colorindex_ci;
    int * dither; /* points to active row of dither matrix */
    int row_index, col_index; /* current indexes into dither matrix */
    int nc = cinfo->out_color_components;
    int ci;
    int row;
    JDIMENSION col;
    JDIMENSION width = cinfo->output_width;

    for (row = 0; row < num_rows; row++) {
        /* Initialize output values to 0 so can process components separately */
        jzero_far((void *) output_buf[row],
                 (size_t) (width * SIZEOF(JSAMPLE)));
        row_index = cquantize->row_index;
        for (ci = 0; ci < nc; ci++) {
            input_ptr = input_buf[row] + ci;
            output_ptr = output_buf[row];
            colorindex_ci = cquantize->colorindex[ci];
            dither = cquantize->odither[ci][row_index];
            col_index = 0;

            for (col = width; col > 0; col--) {
                /* Form pixel value + dither, range-limit to 0..MAXJSAMPLE,
                 * select output value, accumulate into output code for this pixel.
                 * Range-limiting need not be done explicitly, as we have extended
                 * the colorindex table to produce the right answers for out-of-range
                 * inputs. The maximum dither is +- MAXJSAMPLE; this sets the
                 * required amount of padding.
                 */
                *output_ptr += colorindex_ci[GETJSAMPLE(*input_ptr)+dither[col_index]];
                input_ptr += nc;
                output_ptr++;
                col_index = (col_index + 1) & ODITHER_MASK;
            }
            /* Advance row index for next row */
            row_index = (row_index + 1) & ODITHER_MASK;
            cquantize->row_index = row_index;
        }
    }
}

```

```

)

METHODDEF(void)
quantize3_ord_dither (j_decompress_ptr cinfo, JSAMPARRAY input_buf,
                      JSAMPARRAY output_buf, int num_rows)
/* Fast path for out_color_components==3, with ordered dithering */
{
    my_cquantize_ptr cquantize = (my_cquantize_ptr) cinfo->cquantize;
    register int pixcode;
    register JSAMPROW input_ptr;
    register JSAMPROW output_ptr;
    JSAMPROW colorindex0 = cquantize->colorindex[0];
    JSAMPROW colorindex1 = cquantize->colorindex[1];
    JSAMPROW colorindex2 = cquantize->colorindex[2];
    int * dither0;      /* points to active row of dither matrix */
    int * dither1;
    int * dither2;
    int row_index, col_index; /* current indexes into dither matrix */
    int row;
    JDIMENSION col;
    JDIMENSION width = cinfo->output_width;

    for (row = 0; row < num_rows; row++) {
        row_index = cquantize->row_index;
        input_ptr = input_buf[row];
        output_ptr = output_buf[row];
        dither0 = cquantize->odither[0][row_index];
        dither1 = cquantize->odither[1][row_index];
        dither2 = cquantize->odither[2][row_index];
        col_index = 0;

        for (col = width; col > 0; col--) {
            pixcode = GETJSAMPLE(colorindex0[GETJSAMPLE(*input_ptr++) +
                                         dither0[col_index]]);
            pixcode += GETJSAMPLE(colorindex1[GETJSAMPLE(*input_ptr++) +
                                         dither1[col_index]]);
            pixcode += GETJSAMPLE(colorindex2[GETJSAMPLE(*input_ptr++) +
                                         dither2[col_index]]);
            *output_ptr++ = (JSAMPLE) pixcode;
            col_index = (col_index + 1) & ODITHER_MASK;
        }
        row_index = (row_index + 1) & ODITHER_MASK;
        cquantize->row_index = row_index;
    }
}

```

```

METHODDEF(void)
quantize_fs_dither (j_decompress_ptr cinfo, JSAMPARRAY input_buf,
                    JSAMPARRAY output_buf, int num_rows)
/* General case, with Floyd-Steinberg dithering */
{
    my_cquantize_ptr cquantize = (my_cquantize_ptr) cinfo->cquantize;
    register LOCFSEERROR cur; /* current error or pixel value */
    LOCFSEERROR belowerr; /* error for pixel below cur */
    LOCFSEERROR bpreverr; /* error for below/prev col */
    LOCFSEERROR bnexterr; /* error for below/next col */
    LOCFSEERROR delta;
    register FSERRPTR errorptr; /* => ferrors[] at column before current */
    register JSAMPROW input_ptr;
    register JSAMPROW output_ptr;
    JSAMPROW colorindex_ci;
    JSAMPROW colormap_ci;
    int pixcode;
    int nc = cinfo->out_color_components;
    int dir; /* 1 for left-to-right, -1 for right-to-left */
    int dirnc; /* dir * nc */
    int ci;
    int row;
    JDIMENSION col;
    JDIMENSION width = cinfo->output_width;
    JSAMPLE *range_limit = cinfo->sample_range_limit;
    SHIFT_TEMPS

    for (row = 0; row < num_rows; row++) {
        /* Initialize output values to 0 so can process components separately */
        jzero_far((void *) output_buf[row],
                  (size_t) (width * SIZEOF(JSAMPLE)));
        for (ci = 0; ci < nc; ci++) {

```

```

    input_ptr = input_buf[rc + ci;
    output_ptr = output_buf[rc + ci];
    if (cquantize->on_odd_row) {
/* work right to left in this row */
input_ptr += (width-1) * nc; /* so point to rightmost pixel */
output_ptr += width-1;
dir = -1;
dirnc = -nc;
errorptr = cquantize->ferrors[ci] + (width+1); /* => entry after last column */
    } else {
/* work left to right in this row */
dir = 1;
dirnc = nc;
errorptr = cquantize->ferrors[ci]; /* => entry before first column */
    }
    colorindex_ci = cquantize->colorindex[ci];
    colormap_ci = cquantize->sv_colormap[ci];
    /* Preset error values: no error propagated to first pixel from left */
    cur = 0;
    /* and no error propagated to row below yet */
    belowerr = bpreverr = 0;

    for (col = width; col > 0; col--) {
/* cur holds the error propagated from the previous pixel on the
 * current line. Add the error propagated from the previous line
 * to form the complete error correction term for this pixel, and
 * round the error term (which is expressed * 16) to an integer.
 * RIGHT_SHIFT rounds towards minus infinity, so adding 8 is correct
 * for either sign of the error value.
 * Note: errorptr points to *previous* column's array entry.
 */
cur = RIGHT_SHIFT(cur + errorptr[dir] + 8, 4);
/* Form pixel value + error, and range-limit to 0..MAXJSAMPLE.
 * The maximum error is +- MAXJSAMPLE; this sets the required size
 * of the range_limit array.
 */
cur += GETJSAMPLE(*input_ptr);
cur = GETJSAMPLE(range_limit[cur]);
/* Select output value, accumulate into output code for this pixel */
pixcode = GETJSAMPLE(colorindex_ci[cur]);
*output_ptr += (JSAMPLE) pixcode;
/* Compute actual representation error at this pixel */
/* Note: we can do this even though we don't have the final */
/* pixel code, because the colormap is orthogonal. */
cur -= GETJSAMPLE(colormap_ci[pixcode]);
/* Compute error fractions to be propagated to adjacent pixels.
 * Add these into the running sums, and simultaneously shift the
 * next-line error sums left by 1 column.
 */
bnexterr = cur;
delta = cur * 2;
cur += delta; /* form error * 3 */
errorptr[0] = (FSERROR) (bpreverr + cur);
cur += delta; /* form error * 5 */
bpreverr = belowerr + cur;
belowerr = bnexterr;
cur += delta; /* form error * 7 */
/* At this point cur contains the 7/16 error value to be propagated
 * to the next pixel on the current line, and all the errors for the
 * next line have been shifted over. We are therefore ready to move on.
 */
input_ptr += dirnc; /* advance input ptr to next column */
output_ptr += dir; /* advance output ptr to next column */
errorptr += dir; /* advance errorptr to current column */
    }
    /* Post-loop cleanup: we must unload the final error value into the
     * final ferrors[] entry. Note we need not unload belowerr because
     * it is for the dummy column before or after the actual array.
     */
    errorptr[0] = (FSERROR) bpreverr; /* unload prev err into array */
}
cquantize->on_odd_row = (cquantize->on_odd_row ? FALSE : TRUE);
}
}

/*
 * Allocate workspace for Floyd-Steinberg errors.
 */

```

```

LOCAL(void)
alloc_fs_workspace (j_decompress_ptr cinfo)
{
    my_cquantize_ptr cquantize = (my_cquantize_ptr) cinfo->cquantize;
    size_t arraysize;
    int i;

    arraysize = (size_t) ((cinfo->output_width + 2) * SIZEOF(FSERROR));
    for (i = 0; i < cinfo->out_color_components; i++) {
        cquantize->fserrors[i] = (FSERRPTR)
            (*cinfo->mem->alloc_large)((j_common_ptr) cinfo, JPOOL_IMAGE, arraysize);
    }
}

/*
 * Initialize for one-pass color quantization.
 */

METHODDEF(void)
start_pass_1_quant (j_decompress_ptr cinfo, boolean is_pre_scan)
{
    my_cquantize_ptr cquantize = (my_cquantize_ptr) cinfo->cquantize;
    size_t arraysize;
    int i;

    /* Install my colormap. */
    cinfo->colormap = cquantize->sv_colormap;
    cinfo->actual_number_of_colors = cquantize->sv_actual;

    /* Initialize for desired dithering mode. */
    switch (cinfo->dither_mode) {
        case JDITHER_NONE:
            if (cinfo->out_color_components == 3)
                cquantize->pub.color_quantize = color_quantize3;
            else
                cquantize->pub.color_quantize = color_quantize;
            break;
        case JDITHER_ORDERED:
            if (cinfo->out_color_components == 3)
                cquantize->pub.color_quantize = quantize3_ord_dither;
            else
                cquantize->pub.color_quantize = quantize_ord_dither;
            cquantize->row_index = 0; /* initialize state for ordered dither */
            /* If user changed to ordered dither from another mode,
             * we must recreate the color index table with padding.
             * This will cost extra space, but probably isn't very likely.
             */
            if (! cquantize->is_padded)
                create_colorindex(cinfo);
            /* Create ordered-dither tables if we didn't already. */
            if (cquantize->odither[0] == NULL)
                create_odither_tables(cinfo);
            break;
        case JDITHER_FS:
            cquantize->pub.color_quantize = quantize_fs_dither;
            cquantize->on_odd_row = FALSE; /* initialize state for F-S dither */
            /* Allocate Floyd-Steinberg workspace if didn't already. */
            if (cquantize->fserrors[0] == NULL)
                alloc_fs_workspace(cinfo);
            /* Initialize the propagated errors to zero. */
            arraysize = (size_t) ((cinfo->output_width + 2) * SIZEOF(FSERROR));
            for (i = 0; i < cinfo->out_color_components; i++)
                jzero_far((void *) cquantize->fserrors[i], arraysize);
            break;
        default:
            ERREXIT(cinfo, JERR_NOT_COMPILED);
            break;
    }
}

/*
 * Finish up at the end of the pass.
 */

METHODDEF(void)
finish_pass_1_quant (j_decompress_ptr cinfo)
{
    /* no work in 1-pass case */
}

```

```

)

/*
 * Switch to a new external colormap between output passes.
 * Shouldn't get to this module!
 */

METHODDEF(void)
new_color_map_1_quant (j_decompress_ptr cinfo)
{
    ERREXIT(cinfo, JERR_MODE_CHANGE);
}

/*
 * Module initialization routine for 1-pass color quantization.
 */

GLOBAL(void)
jinit_1pass_quantizer (j_decompress_ptr cinfo)
{
    my_cquantize_ptr cquantize;

    cquantize = (my_cquantize_ptr)
        (*cinfo->mem->alloc_small) ((j_common_ptr) cinfo, JPOOL_IMAGE,
            SIZEOF(my_cquantizer));
    cinfo->cquantize = (struct jpeg_color_quantizer *) cquantize;
    cquantize->pub.start_pass = start_pass_1_quant;
    cquantize->pub.finish_pass = finish_pass_1_quant;
    cquantize->pub.new_color_map = new_color_map_1_quant;
    cquantize->fserrors[0] = NULL; /* Flag FS workspace not allocated */
    cquantize->odither[0] = NULL; /* Also flag odither arrays not allocated */

    /* Make sure my internal arrays won't overflow */
    if (cinfo->out_color_components > MAX_Q_COMPS)
        ERREXIT1(cinfo, JERR_QUANT_COMPONENTS, MAX_Q_COMPS);
    /* Make sure colormap indexes can be represented by JSAMPLEs */
    if (cinfo->desired_number_of_colors > (MAXJSAMPLE+1))
        ERREXIT1(cinfo, JERR_QUANT_MANY_COLORS, MAXJSAMPLE+1);

    /* Create the colormap and color index table. */
    create_colormap(cinfo);
    create_colorindex(cinfo);

    /* Allocate Floyd-Steinberg workspace now if requested.
     * We do this now since it is FAR storage and may affect the memory
     * manager's space calculations. If the user changes to FS dither
     * mode in a later pass, we will allocate the space then, and will
     * possibly overrun the max_memory_to_use setting.
     */
    if (cinfo->dither_mode == JDITHER_FS)
        alloc_fs_workspace(cinfo);
}

#endif /* QUANT_1PASS_SUPPORTED */

```

```

/*
 * jquant2.c
 *
 * Copyright (C) 1991-1996, Thomas G. Lane.
 * This file is part of the Independent JPEG Group's software.
 * For conditions of distribution and use, see the accompanying README file.
 *
 * This file contains 2-pass color quantization (color mapping) routines.
 * These routines provide selection of a custom color map for an image,
 * followed by mapping of the image to that color map, with optional
 * Floyd-Steinberg dithering.
 * It is also possible to use just the second pass to map to an arbitrary
 * externally-given color map.
 *
 * Note: ordered dithering is not supported, since there isn't any fast
 * way to compute intercolor distances; it's unclear that ordered dither's
 * fundamental assumptions even hold with an irregularly spaced color map.
 */

#define JPEG_INTERNALS
#include "jinclude.h"
#include "jpeglib.h"

#ifdef QUANT_2PASS_SUPPORTED

/*
 * This module implements the well-known Heckbert paradigm for color
 * quantization. Most of the ideas used here can be traced back to
 * Heckbert's seminal paper
 * Heckbert, Paul. "Color Image Quantization for Frame Buffer Display",
 * Proc. SIGGRAPH '82, Computer Graphics v.16 #3 (July 1982), pp 297-304.
 *
 * In the first pass over the image, we accumulate a histogram showing the
 * usage count of each possible color. To keep the histogram to a reasonable
 * size, we reduce the precision of the input; typical practice is to retain
 * 5 or 6 bits per color, so that 8 or 4 different input values are counted
 * in the same histogram cell.
 *
 * Next, the color-selection step begins with a box representing the whole
 * color space, and repeatedly splits the "largest" remaining box until we
 * have as many boxes as desired colors. Then the mean color in each
 * remaining box becomes one of the possible output colors.
 *
 * The second pass over the image maps each input pixel to the closest output
 * color (optionally after applying a Floyd-Steinberg dithering correction).
 * This mapping is logically trivial, but making it go fast enough requires
 * considerable care.
 *
 * Heckbert-style quantizers vary a good deal in their policies for choosing
 * the "largest" box and deciding where to cut it. The particular policies
 * used here have proved out well in experimental comparisons, but better ones
 * may yet be found.
 *
 * In earlier versions of the IJG code, this module quantized in YCbCr color
 * space, processing the raw upsampled data without a color conversion step.
 * This allowed the color conversion math to be done only once per colormap
 * entry, not once per pixel. However, that optimization precluded other
 * useful optimizations (such as merging color conversion with upsampling)
 * and it also interfered with desired capabilities such as quantizing to an
 * externally-supplied colormap. We have therefore abandoned that approach.
 * The present code works in the post-conversion color space, typically RGB.
 *
 * To improve the visual quality of the results, we actually work in scaled
 * RGB space, giving G distances more weight than R, and R in turn more than
 * B. To do everything in integer math, we must use integer scale factors.
 * The 2/3/1 scale factors used here correspond loosely to the relative
 * weights of the colors in the NTSC grayscale equation.
 * If you want to use this code to quantize a non-RGB color space, you'll
 * probably need to change these scale factors.
 */

#define R_SCALE 2      /* scale R distances by this much */
#define G_SCALE 3      /* scale G distances by this much */
#define B_SCALE 1      /* and B by this much */

/* Relabel R/G/B as components 0/1/2, respecting the RGB ordering defined
 * in jmorecfg.h. As the code stands, it will do the right thing for R,G,B
 * and B,G,R orders. If you define some other weird order in jmorecfg.h,
 * you'll get compile errors until you extend this logic. In that case

```



```

* you'll probably want to twiddle the histogram sizes too.
*/

#if RGB_RED == 0
#define C0_SCALE R_SCALE
#endif
#if RGB_BLUE == 0
#define C0_SCALE B_SCALE
#endif
#if RGB_GREEN == 1
#define C1_SCALE G_SCALE
#endif
#if RGB_RED == 2
#define C2_SCALE R_SCALE
#endif
#if RGB_BLUE == 2
#define C2_SCALE B_SCALE
#endif

/*
 * First we have the histogram data structure and routines for creating it.
 *
 * The number of bits of precision can be adjusted by changing these symbols.
 * We recommend keeping 6 bits for G and 5 each for R and B.
 * If you have plenty of memory and cycles, 6 bits all around gives marginally
 * better results; if you are short of memory, 5 bits all around will save
 * some space but degrade the results.
 * To maintain a fully accurate histogram, we'd need to allocate a "long"
 * (preferably unsigned long) for each cell. In practice this is overkill;
 * we can get by with 16 bits per cell. Few of the cell counts will overflow,
 * and clamping those that do overflow to the maximum value will give close-
 * enough results. This reduces the recommended histogram size from 256Kb
 * to 128Kb, which is a useful savings on PC-class machines.
 * (In the second pass the histogram space is re-used for pixel mapping data;
 * in that capacity, each cell must be able to store zero to the number of
 * desired colors. 16 bits/cell is plenty for that too.)
 * Since the JPEG code is intended to run in small memory model on 80x86
 * machines, we can't just allocate the histogram in one chunk. Instead
 * of a true 3-D array, we use a row of pointers to 2-D arrays. Each
 * pointer corresponds to a C0 value (typically 2^5 = 32 pointers) and
 * each 2-D array has 2^6*2^5 = 2048 or 2^6*2^6 = 4096 entries. Note that
 * on 80x86 machines, the pointer row is in near memory but the actual
 * arrays are in far memory (same arrangement as we use for image arrays).
 */
#define MAXNUMCOLORS (MAXJSAMPLE+1) /* maximum size of colormap */

/* These will do the right thing for either R,G,B or B,G,R color order,
 * but you may not like the results for other color orders.
 */
#define HIST_C0_BITS 5 /* bits of precision in R/B histogram */
#define HIST_C1_BITS 6 /* bits of precision in G histogram */
#define HIST_C2_BITS 5 /* bits of precision in B/R histogram */

/* Number of elements along histogram axes. */
#define HIST_C0_ELEMS (1<<HIST_C0_BITS)
#define HIST_C1_ELEMS (1<<HIST_C1_BITS)
#define HIST_C2_ELEMS (1<<HIST_C2_BITS)

/* These are the amounts to shift an input value to get a histogram index. */
#define C0_SHIFT (BITS_IN_JSAMPLE-HIST_C0_BITS)
#define C1_SHIFT (BITS_IN_JSAMPLE-HIST_C1_BITS)
#define C2_SHIFT (BITS_IN_JSAMPLE-HIST_C2_BITS)

typedef UINT16 histcell; /* histogram cell; prefer an unsigned type */

typedef histcell * histptr; /* for pointers to histogram cells */

typedef histcell hist1d[HIST_C2_ELEMS]; /* typedefs for the array */
typedef hist1d * hist2d; /* type for the 2nd-level pointers */
typedef hist2d * hist3d; /* type for top-level pointer */

/* Declarations for Floyd-Steinberg dithering.
 *
 * Errors are accumulated into the array ferrors[], at a resolution of
 * 1/16th of a pixel count. The error at a given pixel is propagated
 * to its not-yet-processed neighbors using the standard F-S fractions,

```

```

*      ... (here) 7/16
*      3/16      5/16      1/16
* We work left-to-right on even rows, right-to-left on odd rows.
*
* We can get away with a single array (holding one row's worth of errors)
* by using it to store the current row's errors at pixel columns not yet
* processed, but the next row's errors at columns already processed. We
* need only a few extra variables to hold the errors immediately around the
* current column. (If we are lucky, those variables are in registers, but
* even if not, they're probably cheaper to access than array elements are.)
*
* The fserrors[] array has (#columns + 2) entries; the extra entry at
* each end saves us from special-casing the first and last pixels.
* Each entry is three values long, one value for each color component.
*
* Note: on a wide image, we might not have enough room in a PC's near data
* segment to hold the error array; so it is allocated with alloc_large.
*/

#if BITS_IN_JSAMPLE == 8
typedef INT16 FSERROR;      /* 16 bits should be enough */
typedef int LOCFSError;     /* use 'int' for calculation temps */
#else
typedef INT32 FSERROR;      /* may need more than 16 bits */
typedef INT32 LOCFSError;   /* be sure calculation temps are big enough */
#endif

typedef FSERROR *FSERRPTR; /* pointer to error array (in FAR storage!) */

/* Private subobject */
typedef struct {
    struct jpeg_color_quantizer pub; /* public fields */

    /* Space for the eventually created colormap is stashed here */
    JSAMPARRAY sv_colormap; /* colormap allocated at init time */
    int desired; /* desired # of colors = size of colormap */

    /* Variables for accumulating image statistics */
    hist3d histogram; /* pointer to the histogram */

    boolean needs_zeroed; /* TRUE if next pass must zero histogram */

    /* Variables for Floyd-Steinberg dithering */
    FSERRPTR fserrors; /* accumulated errors */
    boolean on_odd_row; /* flag to remember which row we are on */
    int * error_limiter; /* table for clamping the applied error */
    my_cquantizer;

    typedef my_cquantizer * my_cquantize_ptr;

    /*
     * Prescan some rows of pixels.
     * In this module the prescan simply updates the histogram, which has been
     * initialized to zeroes by start_pass.
     * An output_buf parameter is required by the method signature, but no data
     * is actually output (in fact the buffer controller is probably passing a
     * NULL pointer).
     */
}

METHODDEF(void)
prescan_quantize (j_decompress_ptr cinfo, JSAMPARRAY input_buf,
                  JSAMPARRAY output_buf, int num_rows)
{
    my_cquantize_ptr cquantize = (my_cquantize_ptr) cinfo->cquantize;
    register JSAMPROW ptr;
    register histptr histp;
    register hist3d histogram = cquantize->histogram;
    int row;
    JDIMENSION col;
    JDIMENSION width = cinfo->output_width;

    for (row = 0; row < num_rows; row++) {
        ptr = input_buf[row];
        for (col = width; col > 0; col--) {
            /* get pixel value and index into the histogram */
            histp = & histogram[GETJSAMPLE(ptr[0]) >> C0_SHIFT]
                [GETJSAMPLE(ptr[1]) >> C1_SHIFT]

```

```

        [GETJSAMPLE(ptr[0] >> C2_SHIFT)];
/* increment, check for overflow and undo increment if so.
if (++(*histp) <= 0)
(*histp)--;
ptr += 3;
}
)
)

/*
 * Next we have the really interesting routines: selection of a colormap
 * given the completed histogram.
 * These routines work with a list of "boxes", each representing a rectangular
 * subset of the input color space (to histogram precision).
 */

typedef struct {
/* The bounds of the box (inclusive); expressed as histogram indexes */
int c0min, c0max;
int c1min, c1max;
int c2min, c2max;
/* The volume (actually 2-norm) of the box */
INT32 volume;
/* The number of nonzero histogram cells within this box */
long colorcount;
} box;

typedef box * boxptr;

LOCAL(boxptr)
find_biggest_color_pop (boxptr boxlist, int numboxes)
/* Find the splittable box with the largest color population */
/* Returns NULL if no splittable boxes remain */
{
register boxptr boxp;
register int i;
register long maxc = 0;
boxptr which = NULL;

for (i = 0, boxp = boxlist; i < numboxes; i++, boxp++) {
if (boxp->colorcount > maxc && boxp->volume > 0) {
which = boxp;
maxc = boxp->colorcount;
}
}
return which;
}

LOCAL(boxptr)
find_biggest_volume (boxptr boxlist, int numboxes)
/* Find the splittable box with the largest (scaled) volume */
/* Returns NULL if no splittable boxes remain */
{
register boxptr boxp;
register int i;
register INT32 maxv = 0;
boxptr which = NULL;

for (i = 0, boxp = boxlist; i < numboxes; i++, boxp++) {
if (boxp->volume > maxv) {
which = boxp;
maxv = boxp->volume;
}
}
return which;
}

LOCAL(void)
update_box (j_decompress_ptr cinfo, boxptr boxp)
/* Shrink the min/max bounds of a box to enclose only nonzero elements, */
/* and recompute its volume and population */
{
my_cquantize_ptr cquantize = (my_cquantize_ptr) cinfo->cquantize;
hist3d histogram = cquantize->histogram;
histptr histp;
int c0,c1,c2;

```

```

int c0min,c0max,c1min,c1max,c2min,c2max;
INT32 dist0,dist1,dist2;
long ccount;

c0min = boxp->c0min;  c0max = boxp->c0max;
c1min = boxp->c1min;  c1max = boxp->c1max;
c2min = boxp->c2min;  c2max = boxp->c2max;

if (c0max > c0min)
  for (c0 = c0min; c0 <= c0max; c0++)
    for (c1 = c1min; c1 <= c1max; c1++) {
      histp = & histogram[c0][c1][c2min];
      for (c2 = c2min; c2 <= c2max; c2++)
        if (*histp++ != 0) {
          boxp->c0min = c0min = c0;
          goto have_c0min;
        }
    }
have_c0min:
if (c0max > c0min)
  for (c0 = c0max; c0 >= c0min; c0--)
    for (c1 = c1min; c1 <= c1max; c1++) {
      histp = & histogram[c0][c1][c2min];
      for (c2 = c2min; c2 <= c2max; c2++)
        if (*histp++ != 0) {
          boxp->c0max = c0max = c0;
          goto have_c0max;
        }
    }
have_c0max:
if (c1max > c1min)
  for (c1 = c1min; c1 <= c1max; c1++)
    for (c0 = c0min; c0 <= c0max; c0++) {
      histp = & histogram[c0][c1][c2min];
      for (c2 = c2min; c2 <= c2max; c2++)
        if (*histp++ != 0) {
          boxp->c1min = c1min = c1;
          goto have_c1min;
        }
    }
have_c1min:
if (c1max > c1min)
  for (c1 = c1max; c1 >= c1min; c1--)
    for (c0 = c0min; c0 <= c0max; c0++) {
      histp = & histogram[c0][c1][c2min];
      for (c2 = c2min; c2 <= c2max; c2++)
        if (*histp++ != 0) {
          boxp->c1max = c1max = c1;
          goto have_c1max;
        }
    }
have_c1max:
if (c2max > c2min)
  for (c2 = c2min; c2 <= c2max; c2++)
    for (c0 = c0min; c0 <= c0max; c0++) {
      histp = & histogram[c0][c1min][c2];
      for (c1 = c1min; c1 <= c1max; c1++, histp += HIST_C2_ELEMS)
        if (*histp != 0) {
          boxp->c2min = c2min = c2;
          goto have_c2min;
        }
    }
have_c2min:
if (c2max > c2min)
  for (c2 = c2max; c2 >= c2min; c2--)
    for (c0 = c0min; c0 <= c0max; c0++) {
      histp = & histogram[c0][c1min][c2];
      for (c1 = c1min; c1 <= c1max; c1++, histp += HIST_C2_ELEMS)
        if (*histp != 0) {
          boxp->c2max = c2max = c2;
          goto have_c2max;
        }
    }
have_c2max:

/* Update box volume.
 * We use 2-norm rather than real volume here; this biases the method
 * against making long narrow boxes, and it has the side benefit that
 * a box is splittable iff norm > 0.
 * Since the differences are expressed in histogram-cell units,

```

```

* we have to shift back to SAMPLE units to get consistent distances;
* after which, we scale according to the selected distance scale factors.
*/
dist0 = ((c0max - c0min) << C0_SHIFT) * C0_SCALE;
dist1 = ((c1max - c1min) << C1_SHIFT) * C1_SCALE;
dist2 = ((c2max - c2min) << C2_SHIFT) * C2_SCALE;
boxp->volume = dist0*dist0 + dist1*dist1 + dist2*dist2;

/* Now scan remaining volume of box and compute population */
ccount = 0;
for (c0 = c0min; c0 <= c0max; c0++)
    for (c1 = c1min; c1 <= c1max; c1++) {
        histp = & histogram[c0][c1][c2min];
        for (c2 = c2min; c2 <= c2max; c2++, histp++)
            if (*histp != 0) {
                ccount++;
            }
    }
boxp->colorcount = ccount;
}

LOCAL(int)
median_cut (j_decompress_ptr cinfo, boxptr boxlist, int numboxes,
            int desired_colors)
/* Repeatedly select and split the largest box until we have enough boxes */
{
    int n, lb;
    int c0, c1, c2, cmax;
    register boxptr b1, b2;

    while (numboxes < desired_colors) {
        /* Select box to split.
         * Current algorithm: by population for first half, then by volume.
         */
        if (numboxes*2 <= desired_colors) {
            b1 = find_biggest_color_pop(boxlist, numboxes);
        } else {
            b1 = find_biggest_volume(boxlist, numboxes);
        }
        if (b1 == NULL) /* no splittable boxes left! */
            break;
        b2 = &boxlist[numboxes]; /* where new box will go */
        /* Copy the color bounds to the new box. */
        b2->c0max = b1->c0max; b2->c1max = b1->c1max; b2->c2max = b1->c2max;
        b2->c0min = b1->c0min; b2->c1min = b1->c1min; b2->c2min = b1->c2min;
        /* Choose which axis to split the box on.
         * Current algorithm: longest scaled axis.
         * See notes in update_box about scaling distances.
         */
        c0 = ((b1->c0max - b1->c0min) << C0_SHIFT) * C0_SCALE;
        c1 = ((b1->c1max - b1->c1min) << C1_SHIFT) * C1_SCALE;
        c2 = ((b1->c2max - b1->c2min) << C2_SHIFT) * C2_SCALE;
        /* We want to break any ties in favor of green, then red, blue last.
         * This code does the right thing for R,G,B or B,G,R color orders only.
         */
        #if RGB_RED == 0
            cmax = c1; n = 1;
            if (c0 > cmax) { cmax = c0; n = 0; }
            if (c2 > cmax) { n = 2; }
        #else
            cmax = c1; n = 1;
            if (c2 > cmax) { cmax = c2; n = 2; }
            if (c0 > cmax) { n = 0; }
        #endif
        /* Choose split point along selected axis, and update box bounds.
         * Current algorithm: split at halfway point.
         * (Since the box has been shrunk to minimum volume,
         * any split will produce two nonempty subboxes.)
         * Note that lb value is max for lower box, so must be < old max.
         */
        switch (n) {
            case 0:
                lb = (b1->c0max + b1->c0min) / 2;
                b1->c0max = lb;
                b2->c0min = lb+1;
                break;
            case 1:
                lb = (b1->c1max + b1->c1min) / 2;
                b1->c1max = lb;

```

```

        b2->c1min = 1b+1;
        break;
    case 2:
        1b = (b1->c2max + b1->c2min) / 2;
        b1->c2max = 1b;
        b2->c2min = 1b+1;
        break;
    }
    /* Update stats for boxes */
    update_box(cinfo, b1);
    update_box(cinfo, b2);
    numboxes++;
}
return numboxes;
}

```

```

LOCAL(void)
compute_color (j_decompress_ptr cinfo, boxptr boxp, int icolor)
/* Compute representative color for a box, put it in colormap[icolor] */
{
    /* Current algorithm: mean weighted by pixels (not colors) */
    /* Note it is important to get the rounding correct! */
    my_cquantize_ptr cquantize = (my_cquantize_ptr) cinfo->cquantize;
    hist3d histogram = cquantize->histogram;
    histptr histp;
    int c0, c1, c2;
    int c0min, c0max, c1min, c1max, c2min, c2max;
    long count;
    long total = 0;
    long c0total = 0;
    long c1total = 0;
    long c2total = 0;

    c0min = boxp->c0min; c0max = boxp->c0max;
    c1min = boxp->c1min; c1max = boxp->c1max;
    c2min = boxp->c2min; c2max = boxp->c2max;

    for (c0 = c0min; c0 <= c0max; c0++)
        for (c1 = c1min; c1 <= c1max; c1++) {
            histp = & histogram[c0][c1][c2min];
            for (c2 = c2min; c2 <= c2max; c2++) {
                if ((count = *histp++) != 0) {
                    total += count;
                    c0total += ((c0 << C0_SHIFT) + ((1<<C0_SHIFT)>>1)) * count;
                    c1total += ((c1 << C1_SHIFT) + ((1<<C1_SHIFT)>>1)) * count;
                    c2total += ((c2 << C2_SHIFT) + ((1<<C2_SHIFT)>>1)) * count;
                }
            }
        }

    cinfo->colormap[0][icolor] = (JSAMPLE) ((c0total + (total>>1)) / total);
    cinfo->colormap[1][icolor] = (JSAMPLE) ((c1total + (total>>1)) / total);
    cinfo->colormap[2][icolor] = (JSAMPLE) ((c2total + (total>>1)) / total);
}

```

```

LOCAL(void)
select_colors (j_decompress_ptr cinfo, int desired_colors)
/* Master routine for color selection */
{
    boxptr boxlist;
    int numboxes;
    int i;

    /* Allocate workspace for box list */
    boxlist = (boxptr) (*cinfo->mem->alloc_small)
        ((j_common_ptr) cinfo, JPOOL_IMAGE, desired_colors * sizeof(box));
    /* Initialize one box containing whole space */
    numboxes = 1;
    boxlist[0].c0min = 0;
    boxlist[0].c0max = MAXJSAMPLE >> C0_SHIFT;
    boxlist[0].c1min = 0;
    boxlist[0].c1max = MAXJSAMPLE >> C1_SHIFT;
    boxlist[0].c2min = 0;
    boxlist[0].c2max = MAXJSAMPLE >> C2_SHIFT;
    /* Shrink it to actually-used volume and set its statistics */
    update_box(cinfo, & boxlist[0]);
    /* Perform median-cut to produce final box list */
    numboxes = median_cut(cinfo, boxlist, numboxes, desired_colors);
}

```

```

/* Compute the representative color for each box, fill colormap
for (i = 0; i < numboxes; i++)
    compute_color(cinfo, & boxlist[i], i);
cinfo->actual_number_of_colors = numboxes;
TRACEMS1(cinfo, 1, JTRC_QUANT_SELECTED, numboxes);
)

```

```

/*
 * These routines are concerned with the time-critical task of mapping input
 * colors to the nearest color in the selected colormap.
 *
 * We re-use the histogram space as an "inverse color map", essentially a
 * cache for the results of nearest-color searches. All colors within a
 * histogram cell will be mapped to the same colormap entry, namely the one
 * closest to the cell's center. This may not be quite the closest entry to
 * the actual input color, but it's almost as good. A zero in the cache
 * indicates we haven't found the nearest color for that cell yet; the array
 * is cleared to zeroes before starting the mapping pass. When we find the
 * nearest color for a cell, its colormap index plus one is recorded in the
 * cache for future use. The pass2 scanning routines call fill_inverse_cmap
 * when they need to use an unfilled entry in the cache.
 *
 * Our method of efficiently finding nearest colors is based on the "locally
 * sorted search" idea described by Heckbert and on the incremental distance
 * calculation described by Spencer W. Thomas in chapter III.1 of Graphics
 * Gems II (James Arvo, ed. Academic Press, 1991). Thomas points out that
 * the distances from a given colormap entry to each cell of the histogram can
 * be computed quickly using an incremental method: the differences between
 * distances to adjacent cells themselves differ by a constant. This allows a
 * fairly fast implementation of the "brute force" approach of computing the
 * distance from every colormap entry to every histogram cell. Unfortunately,
 * it needs a work array to hold the best-distance-so-far for each histogram
 * cell (because the inner loop has to be over cells, not colormap entries).
 * The work array elements have to be INT32s, so the work array would need
 * 256Kb at our recommended precision. This is not feasible in DOS machines.
 *
 * To get around these problems, we apply Thomas' method to compute the
 * nearest colors for only the cells within a small subbox of the histogram.
 * The work array need be only as big as the subbox, so the memory usage
 * problem is solved. Furthermore, we need not fill subboxes that are never
 * referenced in pass2; many images use only part of the color gamut, so a
 * fair amount of work is saved. An additional advantage of this
 * approach is that we can apply Heckbert's locality criterion to quickly
 * eliminate colormap entries that are far away from the subbox; typically
 * three-fourths of the colormap entries are rejected by Heckbert's criterion,
 * and we need not compute their distances to individual cells in the subbox.
 * The speed of this approach is heavily influenced by the subbox size: too
 * small means too much overhead, too big loses because Heckbert's criterion
 * can't eliminate as many colormap entries. Empirically the best subbox
 * size seems to be about 1/512th of the histogram (1/8th in each direction).
 *
 * Thomas' article also describes a refined method which is asymptotically
 * faster than the brute-force method, but it is also far more complex and
 * cannot efficiently be applied to small subboxes. It is therefore not
 * useful for programs intended to be portable to DOS machines. On machines
 * with plenty of memory, filling the whole histogram in one shot with Thomas'
 * refined method might be faster than the present code --- but then again,
 * it might not be any faster, and it's certainly more complicated.
 */

```

```

/* log2(histogram cells in update box) for each axis; this can be adjusted */
#define BOX_C0_LOG (HIST_C0_BITS-3)
#define BOX_C1_LOG (HIST_C1_BITS-3)
#define BOX_C2_LOG (HIST_C2_BITS-3)

#define BOX_C0_ELEMS (1<<BOX_C0_LOG) /* # of hist cells in update box */
#define BOX_C1_ELEMS (1<<BOX_C1_LOG)
#define BOX_C2_ELEMS (1<<BOX_C2_LOG)

#define BOX_C0_SHIFT (C0_SHIFT + BOX_C0_LOG)
#define BOX_C1_SHIFT (C1_SHIFT + BOX_C1_LOG)
#define BOX_C2_SHIFT (C2_SHIFT + BOX_C2_LOG)

```

```

/*
 * The next three routines implement inverse colormap filling. They could
 * all be folded into one big routine, but splitting them up this way saves
 * some stack space (the mindist[] and bestdist[] arrays need not coexist)

```

```

* and may allow some compilers to produce better code by registering more
* inner-loop variables.
*/

```

```

LOCAL(int)
find_nearby_colors (j_decompress_ptr cinfo, int minc0, int minc1, int minc2,
                    JSAMPLE colorlist[])
/* Locate the colormap entries close enough to an update box to be candidates
 * for the nearest entry to some cell(s) in the update box. The update box
 * is specified by the center coordinates of its first cell. The number of
 * candidate colormap entries is returned, and their colormap indexes are
 * placed in colorlist[].
 * This routine uses Heckbert's "locally sorted search" criterion to select
 * the colors that need further consideration.
 */
{
    int numcolors = cinfo->actual_number_of_colors;
    int maxc0, maxc1, maxc2;
    int centerc0, centerc1, centerc2;
    int i, x, ncolors;
    INT32 minmaxdist, min_dist, max_dist, tdist;
    INT32 mindist[MAXNUMCOLORS]; /* min distance to colormap entry i */

    /* Compute true coordinates of update box's upper corner and center.
     * Actually we compute the coordinates of the center of the upper-corner
     * histogram cell, which are the upper bounds of the volume we care about.
     * Note that since ">" rounds down, the "center" values may be closer to
     * min than to max; hence comparisons to them must be "<=", not "<".
     */
    maxc0 = minc0 + ((1 << BOX_C0_SHIFT) - (1 << C0_SHIFT));
    centerc0 = (minc0 + maxc0) >> 1;
    maxc1 = minc1 + ((1 << BOX_C1_SHIFT) - (1 << C1_SHIFT));
    centerc1 = (minc1 + maxc1) >> 1;
    maxc2 = minc2 + ((1 << BOX_C2_SHIFT) - (1 << C2_SHIFT));
    centerc2 = (minc2 + maxc2) >> 1;

    /* For each color in colormap, find:
     * 1. its minimum squared-distance to any point in the update box
     *    (zero if color is within update box);
     * 2. its maximum squared-distance to any point in the update box.
     * Both of these can be found by considering only the corners of the box.
     * We save the minimum distance for each color in mindist[];
     * only the smallest maximum distance is of interest.
     */
    minmaxdist = 0x7FFFFFFF;

    for (i = 0; i < numcolors; i++) {
        /* We compute the squared-c0-distance term, then add in the other two. */
        x = GETJSAMPLE(cinfo->colormap[0][i]);
        if (x < minc0) {
            tdist = (x - minc0) * C0_SCALE;
            min_dist = tdist*tdist;
            tdist = (x - maxc0) * C0_SCALE;
            max_dist = tdist*tdist;
        } else if (x > maxc0) {
            tdist = (x - maxc0) * C0_SCALE;
            min_dist = tdist*tdist;
            tdist = (x - minc0) * C0_SCALE;
            max_dist = tdist*tdist;
        } else {
            /* within cell range so no contribution to min_dist */
            min_dist = 0;
            if (x <= centerc0) {
                tdist = (x - maxc0) * C0_SCALE;
                max_dist = tdist*tdist;
            } else {
                tdist = (x - minc0) * C0_SCALE;
                max_dist = tdist*tdist;
            }
        }

        x = GETJSAMPLE(cinfo->colormap[1][i]);
        if (x < minc1) {
            tdist = (x - minc1) * C1_SCALE;
            min_dist += tdist*tdist;
            tdist = (x - maxc1) * C1_SCALE;
            max_dist += tdist*tdist;
        } else if (x > maxc1) {
            tdist = (x - maxc1) * C1_SCALE;
            min_dist += tdist*tdist;

```



```

    tdist = (x - minc1) * C1_SCALE;
    max_dist += tdist*tdist;
} else {
    /* within cell range so no contribution to min_dist */
    if (x <= centerc1) {
        tdist = (x - maxc1) * C1_SCALE;
        max_dist += tdist*tdist;
    } else {
        tdist = (x - minc1) * C1_SCALE;
        max_dist += tdist*tdist;
    }
}

x = GETJSAMPLE(cinfo->colormap[2][i]);
if (x < minc2) {
    tdist = (x - minc2) * C2_SCALE;
    min_dist += tdist*tdist;
    tdist = (x - maxc2) * C2_SCALE;
    max_dist += tdist*tdist;
} else if (x > maxc2) {
    tdist = (x - maxc2) * C2_SCALE;
    min_dist += tdist*tdist;
    tdist = (x - minc2) * C2_SCALE;
    max_dist += tdist*tdist;
} else {
    /* within cell range so no contribution to min_dist */
    if (x <= centerc2) {
        tdist = (x - maxc2) * C2_SCALE;
        max_dist += tdist*tdist;
    } else {
        tdist = (x - minc2) * C2_SCALE;
        max_dist += tdist*tdist;
    }
}

mindist[i] = min_dist; /* save away the results */
if (max_dist < minmaxdist)
    minmaxdist = max_dist;
}

/* Now we know that no cell in the update box is more than minmaxdist
 * away from some colormap entry. Therefore, only colors that are
 * within minmaxdist of some part of the box need be considered.
 */
ncolors = 0;
for (i = 0; i < numcolors; i++) {
    if (mindist[i] <= minmaxdist)
        colorlist[ncolors++] = (JSAMPLE) i;
}
return ncolors;
}

LOCAL(void)
find_best_colors (j_decompress_ptr cinfo, int minc0, int minc1, int minc2,
    int numcolors, JSAMPLE colorlist[], JSAMPLE bestcolor[])
/* Find the closest colormap entry for each cell in the update box,
 * given the list of candidate colors prepared by find_nearby_colors.
 * Return the indexes of the closest entries in the bestcolor[] array.
 * This routine uses Thomas' incremental distance calculation method to
 * find the distance from a colormap entry to successive cells in the box.
 */
{
    int ic0, ic1, ic2;
    int i, icolor;
    register INT32 * bptr; /* pointer into bestdist[] array */
    JSAMPLE * cptr; /* pointer into bestcolor[] array */
    INT32 dist0, dist1; /* initial distance values */
    register INT32 dist2; /* current distance in inner loop */
    INT32 xx0, xx1; /* distance increments */
    register INT32 xx2;
    INT32 inc0, inc1, inc2; /* initial values for increments */
    /* This array holds the distance to the nearest-so-far color for each cell */
    INT32 bestdist[BOX_C0_ELEMS * BOX_C1_ELEMS * BOX_C2_ELEMS];

    /* Initialize best-distance for each cell of the update box */
    bptr = bestdist;
    for (i = BOX_C0_ELEMS*BOX_C1_ELEMS*BOX_C2_ELEMS-1; i >= 0; i--)
        *bptr++ = 0x7FFFFFFF;

```

```

/* For each color selected find_nearby_colors,
 * compute its distance to center of each cell in the box.
 * If that's less than best-so-far, update best distance and color number.
 */

/* Nominal steps between cell centers ("x" in Thomas article) */
#define STEP_C0 ((1 << C0_SHIFT) * C0_SCALE)
#define STEP_C1 ((1 << C1_SHIFT) * C1_SCALE)
#define STEP_C2 ((1 << C2_SHIFT) * C2_SCALE)

for (i = 0; i < numcolors; i++) {
    icolor = GETJSAMPLE(colorlist[i]);
    /* Compute (square of) distance from minc0/c1/c2 to this color */
    inc0 = (minc0 - GETJSAMPLE(cinfo->colormap[0][icolor])) * C0_SCALE;
    dist0 = inc0*inc0;
    inc1 = (minc1 - GETJSAMPLE(cinfo->colormap[1][icolor])) * C1_SCALE;
    dist0 += inc1*inc1;
    inc2 = (minc2 - GETJSAMPLE(cinfo->colormap[2][icolor])) * C2_SCALE;
    dist0 += inc2*inc2;
    /* Form the initial difference increments */
    inc0 = inc0 * (2 * STEP_C0) + STEP_C0 * STEP_C0;
    inc1 = inc1 * (2 * STEP_C1) + STEP_C1 * STEP_C1;
    inc2 = inc2 * (2 * STEP_C2) + STEP_C2 * STEP_C2;
    /* Now loop over all cells in box, updating distance per Thomas method */
    bptr = bestdist;
    cptr = bestcolor;
    xx0 = inc0;
    for (ic0 = BOX_C0_ELEMS-1; ic0 >= 0; ic0--) {
        dist1 = dist0;
        xx1 = inc1;
        for (ic1 = BOX_C1_ELEMS-1; ic1 >= 0; ic1--) {
            dist2 = dist1;
            xx2 = inc2;
            for (ic2 = BOX_C2_ELEMS-1; ic2 >= 0; ic2--) {
                if (dist2 < *bptr) {
                    *bptr = dist2;
                    *cptr = (JSAMPLE) icolor;
                }
                dist2 += xx2;
                xx2 += 2 * STEP_C2 * STEP_C2;
                bptr++;
                cptr++;
            }
            dist1 += xx1;
            xx1 += 2 * STEP_C1 * STEP_C1;
        }
        dist0 += xx0;
        xx0 += 2 * STEP_C0 * STEP_C0;
    }
}

LOCAL(void)
fill_inverse_cmap (j_decompress_ptr cinfo, int c0, int c1, int c2)
/* Fill the inverse-colormap entries in the update box that contains */
/* histogram cell c0/c1/c2. (Only that one cell MUST be filled, but */
/* we can fill as many others as we wish.) */
{
    my_cquantize_ptr cquantize = (my_cquantize_ptr) cinfo->cquantize;
    hist3d histogram = cquantize->histogram;
    int minc0, minc1, minc2; /* lower left corner of update box */
    int ic0, ic1, ic2;
    register JSAMPLE * cptr; /* pointer into bestcolor[] array */
    register histptr cachep; /* pointer into main cache array */
    /* This array lists the candidate colormap indexes. */
    JSAMPLE colorlist[MAXNUMCOLORS];
    int numcolors; /* number of candidate colors */
    /* This array holds the actually closest colormap index for each cell. */
    JSAMPLE bestcolor[BOX_C0_ELEMS * BOX_C1_ELEMS * BOX_C2_ELEMS];

    /* Convert cell coordinates to update box ID */
    c0 >>= BOX_C0_LOG;
    c1 >>= BOX_C1_LOG;
    c2 >>= BOX_C2_LOG;

    /* Compute true coordinates of update box's origin corner.
     * Actually we compute the coordinates of the center of the corner
     * histogram cell, which are the lower bounds of the volume we care about.
     */

```

```

minc0 = (c0 << BOX_C0_SHIFT) | ((1 << C0_SHIFT) >> 1);
minc1 = (c1 << BOX_C1_SHIFT) | ((1 << C1_SHIFT) >> 1);
minc2 = (c2 << BOX_C2_SHIFT) | ((1 << C2_SHIFT) >> 1);

/* Determine which colormap entries are close enough to be candidates
 * for the nearest entry to some cell in the update box.
 */
numcolors = find_nearby_colors(cinfo, minc0, minc1, minc2, colorlist);

/* Determine the actually nearest colors. */
find_best_colors(cinfo, minc0, minc1, minc2, numcolors, colorlist,
    bestcolor);

/* Save the best color numbers (plus 1) in the main cache array */
c0 <=< BOX_C0_LOG; /* convert ID back to base cell indexes */
c1 <=< BOX_C1_LOG;
c2 <=< BOX_C2_LOG;
cptr = bestcolor;
for (ic0 = 0; ic0 < BOX_C0_ELEMS; ic0++) {
    for (ic1 = 0; ic1 < BOX_C1_ELEMS; ic1++) {
        cachep = & histogram[c0+ic0][c1+ic1][c2];
        for (ic2 = 0; ic2 < BOX_C2_ELEMS; ic2++) {
            *cachep++ = (histcell) (GETJSAMPLE(*cptr++) + 1);
        }
    }
}

```

Map some rows of pixels to the output colormapped representation.

```

METHODDEF(void)
pass2_no_dither (j_decompress_ptr cinfo,
    JSAMPARRAY input_buf, JSAMPARRAY output_buf, int num_rows)
/* This version performs no dithering */
{
    my_cquantize_ptr cquantize = (my_cquantize_ptr) cinfo->cquantize;
    hist3d histogram = cquantize->histogram;
    register JSAMPROW inptr, outptr;
    register histptr cachep;
    register int c0, c1, c2;
    int row;
    JDIMENSION col;
    JDIMENSION width = cinfo->output_width;

    for (row = 0; row < num_rows; row++) {
        inptr = input_buf[row];
        outptr = output_buf[row];
        for (col = width; col > 0; col--) {
            /* get pixel value and index into the cache */
            c0 = GETJSAMPLE(*inptr++) >> C0_SHIFT;
            c1 = GETJSAMPLE(*inptr++) >> C1_SHIFT;
            c2 = GETJSAMPLE(*inptr++) >> C2_SHIFT;
            cachep = & histogram[c0][c1][c2];
            /* If we have not seen this color before, find nearest colormap entry */
            /* and update the cache */
            if (*cachep == 0)
                fill_inverse_cmap(cinfo, c0, c1, c2);
            /* Now emit the colormap index for this cell */
            *outptr++ = (JSAMPLE) (*cachep - 1);
        }
    }
}

```

```

METHODDEF(void)
pass2_fs_dither (j_decompress_ptr cinfo,
    JSAMPARRAY input_buf, JSAMPARRAY output_buf, int num_rows)
/* This version performs Floyd-Steinberg dithering */
{
    my_cquantize_ptr cquantize = (my_cquantize_ptr) cinfo->cquantize;
    hist3d histogram = cquantize->histogram;
    register LOCFSEERROR cur0, cur1, cur2; /* current error or pixel value */
    LOCFSEERROR belowerr0, belowerr1, belowerr2; /* error for pixel below cur */
    LOCFSEERROR bpreverr0, bpreverr1, bpreverr2; /* error for below/prev col */
    register FSERRPTR errorptr; /* => ferrors[] at column before current */
    JSAMPROW inptr; /* => current input pixel */
    JSAMPROW outptr; /* => current output pixel */

```

```

histptr cachep;
int dir;          /* +1 or -1 depending on direction */
int dir3;         /* 3*dir, for advancing inptr & errorptr */
int row;
JDIMENSION col;
JDIMENSION width = cinfo->output_width;
JSAMPLE *range_limit = cinfo->sample_range_limit;
int *error_limit = cquantize->error_limiter;
JSAMPROW colormap0 = cinfo->colormap[0];
JSAMPROW colormap1 = cinfo->colormap[1];
JSAMPROW colormap2 = cinfo->colormap[2];
SHIFT_TEMPS

for (row = 0; row < num_rows; row++) {
    inptr = input_buf[row];
    outptr = output_buf[row];
    if (cquantize->on_odd_row) {
        /* work right to left in this row */
        inptr += (width-1) * 3; /* so point to rightmost pixel */
        outptr += width-1;
        dir = -1;
        dir3 = -3;
        errorptr = cquantize->fseerrors + (width+1)*3; /* => entry after last column */
        cquantize->on_odd_row = FALSE; /* flip for next time */
    } else {
        /* work left to right in this row */
        dir = 1;
        dir3 = 3;
        errorptr = cquantize->fseerrors; /* => entry before first real column */
        cquantize->on_odd_row = TRUE; /* flip for next time */
    }
    /* Preset error values: no error propagated to first pixel from left */
    cur0 = cur1 = cur2 = 0;
    /* and no error propagated to row below yet */
    belowerr0 = belowerr1 = belowerr2 = 0;
    bpreverr0 = bpreverr1 = bpreverr2 = 0;

    for (col = width; col > 0; col--) {
        /* curN holds the error propagated from the previous pixel on the
         * current line. Add the error propagated from the previous line
         * to form the complete error correction term for this pixel, and
         * round the error term (which is expressed * 16) to an integer.
         * RIGHT_SHIFT rounds towards minus infinity, so adding 8 is correct
         * for either sign of the error value.
         * Note: errorptr points to *previous* column's array entry.
         */
        cur0 = RIGHT_SHIFT(cur0 + errorptr[dir3+0] + 8, 4);
        cur1 = RIGHT_SHIFT(cur1 + errorptr[dir3+1] + 8, 4);
        cur2 = RIGHT_SHIFT(cur2 + errorptr[dir3+2] + 8, 4);
        /* Limit the error using transfer function set by init_error_limit.
         * See comments with init_error_limit for rationale.
         */
        cur0 = error_limit[cur0];
        cur1 = error_limit[cur1];
        cur2 = error_limit[cur2];
        /* Form pixel value + error, and range-limit to 0..MAXJSAMPLE.
         * The maximum error is +- MAXJSAMPLE (or less with error limiting);
         * this sets the required size of the range_limit array.
         */
        cur0 += GETJSAMPLE(inptr[0]);
        cur1 += GETJSAMPLE(inptr[1]);
        cur2 += GETJSAMPLE(inptr[2]);
        cur0 = GETJSAMPLE(range_limit[cur0]);
        cur1 = GETJSAMPLE(range_limit[cur1]);
        cur2 = GETJSAMPLE(range_limit[cur2]);
        /* Index into the cache with adjusted pixel value */
        cachep = & histogram[cur0>>C0_SHIFT][cur1>>C1_SHIFT][cur2>>C2_SHIFT];
        /* If we have not seen this color before, find nearest colormap */
        /* entry and update the cache */
        if (*cachep == 0)
            fill_inverse_cmap(cinfo, cur0>>C0_SHIFT, cur1>>C1_SHIFT, cur2>>C2_SHIFT);
        /* Now emit the colormap index for this cell */
        { register int pixcode = *cachep - 1;
          *outptr = (JSAMPLE) pixcode;
          /* Compute representation error for this pixel */
          cur0 -= GETJSAMPLE(colormap0[pixcode]);
          cur1 -= GETJSAMPLE(colormap1[pixcode]);
          cur2 -= GETJSAMPLE(colormap2[pixcode]);
        }
        /* Compute error fractions to be propagated to adjacent pixels.

```

```

* Add these into the running sums, and simultaneously shift the
* next-line error sum left by 1 column.
*/

```

```

{ register LOCFSEERROR bnxterr, delta;

```

```

bnxterr = cur0; /* Process component 0 */
delta = cur0 * 2;
cur0 += delta; /* form error * 3 */
errorptr[0] = (FSERROR) (bprevrr0 + cur0);
cur0 += delta; /* form error * 5 */
bprevrr0 = belowerr0 + cur0;
belowerr0 = bnxterr;
cur0 += delta; /* form error * 7 */
bnxterr = cur1; /* Process component 1 */
delta = cur1 * 2;
cur1 += delta; /* form error * 3 */
errorptr[1] = (FSERROR) (bprevrr1 + cur1);
cur1 += delta; /* form error * 5 */
bprevrr1 = belowerr1 + cur1;
belowerr1 = bnxterr;
cur1 += delta; /* form error * 7 */
bnxterr = cur2; /* Process component 2 */
delta = cur2 * 2;
cur2 += delta; /* form error * 3 */
errorptr[2] = (FSERROR) (bprevrr2 + cur2);
cur2 += delta; /* form error * 5 */
bprevrr2 = belowerr2 + cur2;
belowerr2 = bnxterr;
cur2 += delta; /* form error * 7 */
}
/* At this point curN contains the 7/16 error value to be propagated
* to the next pixel on the current line, and all the errors for the
* next line have been shifted over. We are therefore ready to move on.
*/
inptr += dir3; /* Advance pixel pointers to next column */
outptr += dir;
errorptr += dir3; /* advance errorptr to current column */
}

```

```

/* Post-loop cleanup: we must unload the final error values into the
* final fserrors[] entry. Note we need not unload belowerrN because
* it is for the dummy column before or after the actual array.
*/

```

```

errorptr[0] = (FSERROR) bprevrr0; /* unload prev errs into array */
errorptr[1] = (FSERROR) bprevrr1;
errorptr[2] = (FSERROR) bprevrr2;

```

Initialize the error-limiting transfer function (lookup table).

```

* The raw F-S error computation can potentially compute error values of up to
* +- MAXJSAMPLE. But we want the maximum correction applied to a pixel to be
* much less, otherwise obviously wrong pixels will be created. (Typical
* effects include weird fringes at color-area boundaries, isolated bright
* pixels in a dark area, etc.) The standard advice for avoiding this problem
* is to ensure that the "corners" of the color cube are allocated as output
* colors; then repeated errors in the same direction cannot cause cascading
* error buildup. However, that only prevents the error from getting
* completely out of hand; Aaron Giles reports that error limiting improves
* the results even with corner colors allocated.
* A simple clamping of the error values to about +- MAXJSAMPLE/8 works pretty
* well, but the smoother transfer function used below is even better. Thanks
* to Aaron Giles for this idea.
*/

```

```

LOCAL(void)

```

```

init_error_limit (j_decompress_ptr cinfo)

```

```

/* Allocate and fill in the error_limiter table */

```

```

{
my_cquantize_ptr cquantize = (my_cquantize_ptr) cinfo->cquantize;
int * table;
int in, out;

table = (int *) (*cinfo->mem->alloc_small)
((j_common_ptr) cinfo, JPOOL_IMAGE, (MAXJSAMPLE*2+1) * sizeof(int));
table += MAXJSAMPLE; /* so can index -MAXJSAMPLE .. +MAXJSAMPLE */
cquantize->error_limiter = table;

```

```

#define STEPSIZE ((MAXJSAMPLE+1)/16)

```

```

/* Map errors 1:1 up to +- MAXJSAMPLE/16 */
out = 0;
for (; in < STEPSIZE; in++, out++) {
    table[in] = out; table[-in] = -out;
}
/* Map errors 1:2 up to +- 3*MAXJSAMPLE/16 */
for (; in < STEPSIZE*3; in++, out += (in&1) ? 0 : 1) {
    table[in] = out; table[-in] = -out;
}
/* Clamp the rest to final out value (which is (MAXJSAMPLE+1)/8) */
for (; in <= MAXJSAMPLE; in++) {
    table[in] = out; table[-in] = -out;
}
#undef STEPSIZE
)

/*
 * Finish up at the end of each pass.
 */

METHODDEF(void)
finish_pass1 (j_decompress_ptr cinfo)
{
    my_cquantize_ptr cquantize = (my_cquantize_ptr) cinfo->cquantize;

    /* Select the representative colors and fill in cinfo->colormap */
    cinfo->colormap = cquantize->sv_colormap;
    select_colors(cinfo, cquantize->desired);
    /* Force next pass to zero the color index table */
    cquantize->needs_zeroed = TRUE;
}

METHODDEF(void)
finish_pass2 (j_decompress_ptr cinfo)
{
    /* no work */
}

/* Initialize for each processing pass.
 */

METHODDEF(void)
start_pass_2_quant (j_decompress_ptr cinfo, boolean is_pre_scan)
{
    my_cquantize_ptr cquantize = (my_cquantize_ptr) cinfo->cquantize;
    hist3d histogram = cquantize->histogram;
    int i;

    /* Only F-S dithering or no dithering is supported. */
    /* If user asks for ordered dither, give him F-S. */
    if (cinfo->dither_mode != JDITHER_NONE)
        cinfo->dither_mode = JDITHER_FS;

    if (is_pre_scan) {
        /* Set up method pointers */
        cquantize->pub.color_quantize = prescan_quantize;
        cquantize->pub.finish_pass = finish_pass1;
        cquantize->needs_zeroed = TRUE; /* Always zero histogram */
    } else {
        /* Set up method pointers */
        if (cinfo->dither_mode == JDITHER_FS)
            cquantize->pub.color_quantize = pass2_fs_dither;
        else
            cquantize->pub.color_quantize = pass2_no_dither;
        cquantize->pub.finish_pass = finish_pass2;

        /* Make sure color count is acceptable */
        i = cinfo->actual_number_of_colors;
        if (i < 1)
            ERREXIT1(cinfo, JERR_QUANT_FEW_COLORS, 1);
        if (i > MAXNUMCOLORS)
            ERREXIT1(cinfo, JERR_QUANT_MANY_COLORS, MAXNUMCOLORS);

        if (cinfo->dither_mode == JDITHER_FS) {
            size_t arraysize = (size_t) ((cinfo->output_width + 2) *
                (3 * SIZEOF(FSERROR)));

```

```

    Allocate Floyd-Steinberg workspace if we didn't already.
    if (cquantize->fserrors == NULL)
        cquantize->fserrors = (FSERR_PTR) (*cinfo->mem->alloc_large)
            ((j_common_ptr) cinfo, JPOOL_IMAGE, arraysize);
    /* Initialize the propagated errors to zero. */
    jzero_far((void *) cquantize->fserrors, arraysize);
    /* Make the error-limit table if we didn't already. */
    if (cquantize->error_limiter == NULL)
        init_error_limit(cinfo);
    cquantize->on_odd_row = FALSE;
}

/* Zero the histogram or inverse color map, if necessary */
if (cquantize->needs_zeroed) {
    for (i = 0; i < HIST_C0_ELEMS; i++) {
        jzero_far((void *) histogram[i],
            HIST_C1_ELEMS*HIST_C2_ELEMS * sizeof(histcell));
    }
    cquantize->needs_zeroed = FALSE;
}

/*
 * Switch to a new external colormap between output passes.
 */

METHODDEF(void)
new_color_map_2_quant (j_decompress_ptr cinfo)
{
    my_cquantize_ptr cquantize = (my_cquantize_ptr) cinfo->cquantize;

    /* Reset the inverse color map */
    cquantize->needs_zeroed = TRUE;
}

/*
 * Module initialization routine for 2-pass color quantization.
 */

GLOBAL(void)
init_2pass_quantizer (j_decompress_ptr cinfo)
{
    my_cquantize_ptr cquantize;
    int i;

    cquantize = (my_cquantize_ptr)
        (*cinfo->mem->alloc_small) ((j_common_ptr) cinfo, JPOOL_IMAGE,
            sizeof(my_cquantizer));
    cinfo->cquantize = (struct jpeg_color_quantizer *) cquantize;
    cquantize->pub.start_pass = start_pass_2_quant;
    cquantize->pub.new_color_map = new_color_map_2_quant;
    cquantize->fserrors = NULL; /* flag optional arrays not allocated */
    cquantize->error_limiter = NULL;

    /* Make sure jdmaster didn't give me a case I can't handle */
    if (cinfo->out_color_components != 3)
        ERREXIT(cinfo, JERR_NOTIMPL);

    /* Allocate the histogram/inverse colormap storage */
    cquantize->histogram = (hist3d) (*cinfo->mem->alloc_small)
        ((j_common_ptr) cinfo, JPOOL_IMAGE, HIST_C0_ELEMS * sizeof(hist2d));
    for (i = 0; i < HIST_C0_ELEMS; i++) {
        cquantize->histogram[i] = (hist2d) (*cinfo->mem->alloc_large)
            ((j_common_ptr) cinfo, JPOOL_IMAGE,
                HIST_C1_ELEMS*HIST_C2_ELEMS * sizeof(histcell));
    }
    cquantize->needs_zeroed = TRUE; /* histogram is garbage now */

    /* Allocate storage for the completed colormap, if required.
     * We do this now since it is FAR storage and may affect
     * the memory manager's space calculations.
     */
    if (cinfo->enable_2pass_quant) {
        /* Make sure color count is acceptable */
        int desired = cinfo->desired_number_of_colors;
        /* Lower bound on # of colors ... somewhat arbitrary as long as > 0 */
        if (desired < 8)

```





```

/*
 * jutil.c
 *
 * Copyright (C) 1991-1996, Thomas G. Lane.
 * This file is part of the Independent JPEG Group's software.
 * For conditions of distribution and use, see the accompanying README file.
 *
 * This file contains tables and miscellaneous utility routines needed
 * for both compression and decompression.
 * Note we prefix all global names with "j" to minimize conflicts with
 * a surrounding application.
 */

#define JPEG_INTERNALS
#include "jinclude.h"
#include "jpeglib.h"

/*
 * jpeg_zigzag_order[i] is the zigzag-order position of the i'th element
 * of a DCT block read in natural order (left to right, top to bottom).
 */

#if 0 /* This table is not actually needed in v6a */

const int jpeg_zigzag_order[DCTSIZE2] = {
  0, 1, 5, 6, 14, 15, 27, 28,
  2, 4, 7, 13, 16, 26, 29, 42,
  3, 8, 12, 17, 25, 30, 41, 43,
  9, 11, 18, 24, 31, 40, 44, 53,
  10, 19, 23, 32, 39, 45, 52, 54,
  20, 22, 33, 38, 46, 51, 55, 60,
  21, 34, 37, 47, 50, 56, 59, 61,
  35, 36, 48, 49, 57, 58, 62, 63
};

#endif

/*
 * jpeg_natural_order[i] is the natural-order position of the i'th element
 * of zigzag order.
 *
 * When reading corrupted data, the Huffman decoders could attempt
 * to reference an entry beyond the end of this array (if the decoded
 * zero run length reaches past the end of the block). To prevent
 * wild stores without adding an inner-loop test, we put some extra
 * "63"s after the real entries. This will cause the extra coefficient
 * to be stored in location 63 of the block, not somewhere random.
 * The worst case would be a run-length of 15, which means we need 16
 * fake entries.
 */
const int jpeg_natural_order[DCTSIZE2+16] = {
  0, 1, 8, 16, 9, 2, 3, 10,
  17, 24, 32, 25, 18, 11, 4, 5,
  12, 19, 26, 33, 40, 48, 41, 34,
  27, 20, 13, 6, 7, 14, 21, 28,
  35, 42, 49, 56, 57, 50, 43, 36,
  29, 22, 15, 23, 30, 37, 44, 51,
  58, 59, 52, 45, 38, 31, 39, 46,
  53, 60, 61, 54, 47, 55, 62, 63,
  63, 63, 63, 63, 63, 63, 63, 63, /* extra entries for safety in decoder */
  63, 63, 63, 63, 63, 63, 63, 63
};

/*
 * Arithmetic utilities
 */

GLOBAL(long)
jdiv_round_up(long a, long b)
/* Compute a/b rounded up to next integer, ie, ceil(a/b) */
/* Assumes a >= 0, b > 0 */
{
  return (a + b - 1L) / b;
}

GLOBAL(long)

```

```

jround_up (long a, long b)
/* Compute a rounded up to nearest multiple of b, ie, ceil(a/b)*b */
/* Assumes a >= 0, b > 0 */
{
    a += b - 1L;
    return a - (a % b);
}

/* On normal machines we can apply MEMCOPY() and MEMZERO() to sample arrays
 * and coefficient-block arrays. This won't work on 80x86 because the arrays
 * are FAR and we're assuming a small-pointer memory model. However, some
 * DOS compilers provide far-pointer versions of memcpy() and memset() even
 * in the small-model libraries. These will be used if USE_FMEM is defined.
 * Otherwise, the routines below do it the hard way. (The performance cost
 * is not all that great, because these routines aren't very heavily used.)
 */

#ifdef NEED_FAR_POINTERS /* normal case, same as regular macros */
#define FMEMCOPY(dest,src,size) MEMCOPY(dest,src,size)
#define FMEMZERO(target,size) MEMZERO(target,size)
#else /* 80x86 case, define if we can */
#ifdef USE_FMEM
#define FMEMCOPY(dest,src,size) _fmemcpy((void *) (dest), (const void *) (src), (size_t) (size))
#define FMEMZERO(target,size) _fmemset((void *) (target), 0, (size_t) (size))
#endif
#endif

GLOBAL(void)
jcopy_sample_rows (JSAMPARRAY input_array, int source_row,
                   JSAMPARRAY output_array, int dest_row,
                   int num_rows, JDIMENSION num_cols)
/* Copy some rows of samples from one place to another.
 * num_rows rows are copied from input_array[source_row++]
 * to output_array[dest_row++]; these areas may overlap for duplication.
 * The source and destination arrays must be at least as wide as num_cols.
 */
{
    register JSAMPROW inptr, outptr;
#ifdef FMEMCOPY
    register size_t count = (size_t) (num_cols * SIZEOF(JSAMPLE));
#else
    register JDIMENSION count;
#endif
    register int row;

    input_array += source_row;
    output_array += dest_row;

    for (row = num_rows; row > 0; row--) {
        inptr = *input_array++;
        outptr = *output_array++;
#ifdef FMEMCOPY
        FMEMCOPY(outptr, inptr, count);
#else
        for (count = num_cols; count > 0; count--)
            *outptr++ = *inptr++; /* needn't bother with GETJSAMPLE() here */
#endif
    }
}

GLOBAL(void)
jcopy_block_row (JBLOCKROW input_row, JBLOCKROW output_row,
                 JDIMENSION num_blocks)
/* Copy a row of coefficient blocks from one place to another. */
{
#ifdef FMEMCOPY
    FMEMCOPY(output_row, input_row, num_blocks * (DCTSIZE2 * SIZEOF(JCOEF)));
#else
    register JCOEFPTR inptr, outptr;
    register long count;

    inptr = (JCOEFPTR) input_row;
    outptr = (JCOEFPTR) output_row;
    for (count = (long) num_blocks * DCTSIZE2; count > 0; count--) {
        *outptr++ = *inptr++;
    }
#endif
}

```

```

)

GLOBAL(v_id)
jzero_far (void * target, size_t bytestozero)
/* Zero out a chunk of FAR memory. */
/* This might be sample-array data, block-array data, or alloc_large data. */
{
#ifdef FMEMZERO
    FMEMZERO(target, bytestozero);
#else
    register char * ptr = (char *) target;
    register size_t count;

    for (count = bytestozero; count > 0; count--) {
        *ptr++ = 0;
    }
#endif
}

```

[illegible]

```

/*
 * @author Imtiaz Hossain
 *
 * @version 2.0
 *
 */

#include <stdlib.h>
#include <malloc.h>
#include <jerror.h>
#include <setjmp.h>
#include "jpeg_class.h"

// static consts
// Error definitions
const int JPEG::SUCCESS=0;
const int JPEG::MEMORY_ALLOC_ERROR=1;
const int JPEG::FILE_READ_ERROR=2;
const int JPEG::FILE_WRITE_ERROR=3;
const int JPEG::JPEG_LIB_STRUCT_INIT_ERROR=4;

// Coding types

const int JPEG::DEFAULT_CODING=0;
const int JPEG::BASELINE=0;
const int JPEG::PROGRESSIVE=1;
const int JPEG::LOSSLESS=2;

// Resolution
const int JPEG::DEFAULT_RES=1;
const int JPEG::ONE_BYTE=1;
const int JPEG::TWO_BYTE=2;
const int JPEG::THREE_BYTE=3;
const int JPEG::FOUR_BYTE=4;

// Color Planes
const int JPEG::DEFAULT_BPP=3;
const int JPEG::Gray=1;
const int JPEG::RGB=3;

// Image Quality
const int JPEG::DEF_QUALITY=60;

// Compression Ratio
const float JPEG::DEF_RATIO=0.50;

// Time limit on the compression
const long DEF_TIME=5; // 5 seconds .

BYTE * JPEG::Encode(BYTE *jpeg_get_Buffer, int *length, int *ret_quality, int n_Height, int n_Width, int n_Bpp, int n_Quality, int n_Res, int n_Coding)
{
    struct jpeg_compress_struct c_struct;
    struct jpeg_error_mgr jerr;
    JSAMPLE * ptr_to_buffer[1];
    int buffer_length, counter=0, i;
    JOCTET * returnbuffer;

    BYTE * jpeg_RGB_Buffer;
    int bpp;
    int res;

    bpp=n_Bpp;
    res=n_Res;

    jpeg_RGB_Buffer=Resolution_Convertor(jpeg_get_Buffer, &bpp, &res, n_Height, &n_Width);
    n_Bpp=bpp;

    n_ImageHeight=n_Height;
    n_ImageWidth=n_Width;
    n_ImageBpp=n_Bpp;

```

```

//*** Need to assert BPP=3, and Width and Height >0
n_ImageResolution=n_Res;
n_ImageCodingType=n_Coding;
n_ImageQuality=n_Quality;

buffer_length=n_ImageWidth*n_ImageBpp;

ptr_to_buffer[0]=(JSAMPLE *)malloc(buffer_length*sizeof(JSAMPLE));

if(ptr_to_buffer[0]==NULL)
{
    fprintf(stderr,"Memory Allocation error!!!\n");
    exit(1);
}

// First Step: Initializing JPEG compression object
// jpeglib : error handling code
c_struct.err = jpeg_std_error(&jerr);

jpeg_create_compress(&c_struct); // object initialization
c_struct.index=0;

// Next Step (2) : We will not be using a File. So skipping source definition.
//      Initializing Height, Width and Bpp properties.
c_struct.image_width = n_ImageWidth;
c_struct.image_height = n_ImageHeight;
if (n_ImageBpp==3)
{
    c_struct.input_components = 3;      // # of color components/Bytes per pixel.
    c_struct.in_color_space = JCS_RGB;  // colorspace for input image.
}
else
{
    c_struct.input_components = 1;      // # of color components/Bytes per pixel.
    c_struct.in_color_space = JCS_GRAYSCALE; // colorspace for input image.
}

jpeg_buffer_dest(&c_struct); // ### new modification. Instead of jpeg_stdio_src.

// Next Step (3) : Setting defaults and any special parameters like Quality, Coding, etc..
jpeg_set_defaults(&c_struct);
jpeg_set_quality(&c_struct, n_ImageQuality, TRUE );

// Next Step (4): Initializing compressor
jpeg_start_compress(&c_struct, TRUE);

// Next Step (5) : Compressing one scanline at a time.
while (c_struct.next_scanline < c_struct.image_height)
{
    for(i=0;i<buffer_length;i++)
    {
        ptr_to_buffer[0][i] = (JSAMPLE)jpeg_RGB_Buffer[counter];
        counter++;
    }

    jpeg_write_scanlines(&c_struct, ptr_to_buffer, 1);
}

// Last Step (6): Clean up
jpeg_finish_compress(&c_struct);

return(buffer=c_struct.dest->outbuffer;
c_struct.dest->outbuffer=NULL;

jpeg_destroy_compress(&c_struct);

free(ptr_to_buffer[0]);

```

```

*length=(int)c_struct.index;
*ret_quality=(int)n_ImageQuality;

free(fp=1_RGB_Buffer);

return (IF **)(returnbuffer);

} // end of method : compute()

// -----
// ##### DECODER
// -----

BYTE * JPEG::Decode(BYTE *CompressedBuffer, int length){

int i,counter=0;
JSAMPLE *raw_buffer;

JSAMPLE *temp_buffer[1]; /* temporary buffer to read in the scanlines. */
int buffer_length;

/* First Step: assigning structure variables */
struct jpeg_decompress_struct d_struct; /* decompression structure */
struct jpeg_error_mgr jerr; /* error handling stuff */
d_struct.err=jpeg_std_error(&jerr);

/* ### error handler ( Will check later) */
/* Next Step: initialize decompression structure variable */
jpeg_create_decompress(&d_struct);

/* ### Getting a pointer in the structure to the Compressed Input Data */
/* Next Step: Store file pointer in decompression structure. */
// jpeg_stdio_src(&d_struct,in_fp);

jpeg_buffer_src(&d_struct);

d_struct.src->binbuffer=(JSAMPLE *)CompressedBuffer;
d_struct.src->buffer_length=length;

/* Next Step: Need the info. from the header to decompress the data */
jpeg_read_header(&d_struct,TRUE); /* ### Need to figure out what the "TRUE" does. */

n_ImageWidth=d_struct.image_width;
n_ImageHeight=d_struct.image_height;
n_ImageBpp=d_struct.num_components;

buffer_length=n_ImageWidth*n_ImageBpp; /* width of buffer to take care of RGB values in succession..
.. */

/* buffers are being allocated */
/* Here goes.... */

temp_buffer[0]=(JSAMPLE *)malloc(buffer_length*sizeof(JSAMPLE));
if (temp_buffer[0]==NULL)
{
printf(stderr,"Out of memory for temporary buffer \n");

```

```

    }

    /* Raw Buffer */
    raw_buffer=(JSAMPLE *)malloc(n_ImageHeight*buffer_length*sizeof(JSAMPLE ));
    if (raw_buffer==NULL)
    {
        fprintf(stderr,"Out of memory for raw buffer \n");
        exit(1);
    }

    /* Next Step: signal start decompression */
    jpeg_start_decompress(&d_struct);

    /* Check on dimensions */
    if (!((n_ImageHeight==(int)d_struct.output_height)&&(n_ImageWidth==(int)d_struct.output_width)&&(n_ImageBpp==(int)d_struct.out_color_components)))
    {
        fprintf(stderr,"Image dimensions / bpp do not match\n");
        exit(1);
    }

    /* Next Step: Do the actual reading... */
    while ((int)d_struct.output_scanline<n_ImageHeight)
    {
        jpeg_read_scanlines(&d_struct,temp_buffer,1);
        for(i=0;i<buffer_length;i++)
        {
            raw_buffer[counter]=temp_buffer[0][i];
            counter++;
        }
    }

    /* end of the while loop.*/
    free(temp_buffer[0]);

    /* Final Step: It finally works!!!
    jpeg_finish_decompress(&d_struct);
    jpeg_destroy_decompress(&d_struct);
    printf("Decoder counter = %d\n",counter);
    // ### remember: free temp_buffer
    return (BYTE *)raw_buffer;

    *//* end of Decode. */

/* Method ChopWindow..... get a central window with 1/7th the original sides. */

BYTE * JPEG::ChopWindow(BYTE * whole_stream, int * height, int * width, int bpp)
{
    int row_start, row_end, col_start, col_end;
    int i, j, s_k=1, w_k=1, diff_r, diff_c, diffcb;
    int ht, wd, wdb, c_start, c_end;

    BYTE * small_buff;

    ht=*height;
    wd=*width;

    row_start=(3*ht)/7;
    row_end=(4*ht)/7;

    diff_r=row_end-row_start+1;

```

```

*height=diffr;

c_start=(row_start-1)*bpp;
c_end=(row_end)*bpp;

diffc=c_end-c_start+1;

col_start=(c_start-1)*bpp+1;
col_end=(c_end)*bpp;

diffcb=diffc*bpp; // multiply after the calculation. :)

*width=diffc;

small_buff=(BYTE *)malloc(diffr*diffcb*sizeof(BYTE));
if (small_buff==NULL)
{
    fprintf(stderr,"Memory Alloc error \n");
    exit(1);
}

wdb=wd*bpp;

for (i=row_start;i<=row_end;i++)
{
    for(j=col_start;j<=col_end;j++)
    {
        if ((i>=row_start)&&(i<=row_end))
        {
            if ((j>=col_start)&&(j<=col_end))
            {
                small_buff[s_k-1]=whole_stream[w_k-1];
                s_k++;
            }
            w_k++;
        }
    }
}

return small_buff;

```

```

/* Overloaded Encode function to take care of compression ratio */

```

```

BYTE * JPEG::Encode(BYTE *jpeg_get_Buffer,int *length, int * ret_quality, int n_Height, int n_Width,
    int n_Bpp, float n_Ratio, long n_Time, int n_Res, int n_Coding)
{
    BYTE *jpeg_SmallRawBuffer; // for the small test window we're working with .
    BYTE *jpeg_DataBuffer;
    int len=0;
    int n_height, n_width, small_img_size;
    int n_store_height,n_store_width;
    float comp_ratio;
    float lo_comp_ratio, hi_comp_ratio, mid_comp_ratio;
    int quality_lo_lmt=10, quality_hi_lmt=90, quality;
    int quality_mid_lmt;
    int flag,iterations=0;
    long max_time, sec_elapsed;

    time_t start_time, end_time;
    double time_span;

    BYTE * jpeg_RGB_Buffer;
    int bpp;
    int res;

```



```

bpp = 1;
ret = 0;

jpeg_RGB_Buffer = Resolution_Convertor(jpeg_get_Buffer, &bpp, &res, n_Height, &n_Width);
n_ImageBpp;

n_ImageHeight = n_Height;
n_ImageWidth = n_Width;
n_ImageBpp = n_Bpp;
// need to assert BPP=1 or 3, and Width and Height >0

n_ImageResolution = n_Res;
n_ImageCodingType = n_Coding;
comp_ratio = n_Ratio; // specifying compression ratio instead of the Quality factor.
max_time = n_Time; // Maximum # of seconds that will be tolerated for the determination of the
optimum quality factor.

// By default I'm taking, say, an n by m window, where n = (1/7)*ImageHeight
// m = (1/7)*ImageWidth*ImageBpp.

quality_mid_lmt = (quality_hi_lmt + quality_lo_lmt) / 2;

n_height = n_ImageHeight;
n_width = n_ImageWidth;

jpeg_SmallRawBuffer = ChopWindow(jpeg_RGB_Buffer, &n_height, &n_width, (int)n_ImageBpp);
small_img_size = n_height * n_width * n_ImageBpp;

n_store_height = n_ImageHeight;
n_store_width = n_ImageWidth;

n_ImageHeight = n_height;
n_ImageWidth = n_width;

// jpeg_DataBuffer = test.Encode(jpeg_RawBuffer, &len);

// to get the upper and lower bound ratios corresponding to the upper and lower
// bounds of the quality factor i.e. 90 and 10 respt.

// for the lower bound on the quality factor i.e. 10.
jpeg_DataBuffer = Encode(jpeg_SmallRawBuffer, &len, ret_quality, (int)n_ImageHeight, (int)n_ImageWidth,
(int)n_ImageBpp, (int)10);
lo_comp_ratio = (float)len / (float)small_img_size;
free(jpeg_DataBuffer);

// for the upper bound on the quality factor i.e. 90.
jpeg_DataBuffer = Encode(jpeg_SmallRawBuffer, &len, ret_quality, (int)n_ImageHeight, (int)n_ImageWidth,
(int)n_ImageBpp, (int)90);
hi_comp_ratio = (float)len / (float)small_img_size;
free(jpeg_DataBuffer);

// for the middle bound on the quality factor i.e. (90-10)/2.
jpeg_DataBuffer = Encode(jpeg_SmallRawBuffer, &len, ret_quality, (int)n_ImageHeight, (int)n_ImageWidth,
(int)n_ImageBpp, (int)quality_mid_lmt);
mid_comp_ratio = (float)len / (float)small_img_size;
free(jpeg_DataBuffer);

// start of determination of quality factor.

flag = 0;
time(&start_time);

while (flag == 0)
{
    if (comp_ratio >= hi_comp_ratio)
    {
        quality = quality_lo_lmt;
    }
}

```

```

    }
    if (comp_ratio <= lo_comp_ratio)
    {
        quality = quality_hi_lmt;
        flag = 1;
    }

    if (comp_ratio < mid_comp_ratio) && (comp_ratio > lo_comp_ratio)
    {
        quality_hi_lmt = quality_mid_lmt;
        mid_comp_ratio = mid_comp_ratio;
        quality_mid_lmt = (quality_hi_lmt + quality_lo_lmt) / 2;

        jpeg_DataBuffer = Encode(jpeg_SmallRawBuffer, &len, ret_quality, (int)n_ImageHeight, (int)n_ImageWidth, (int)n_ImageBpp, (int)quality_mid_lmt);
        mid_comp_ratio = (float)len / (float)small_img_size;
        free(jpeg_DataBuffer);
    }
    else
    {
        if (comp_ratio >= mid_comp_ratio) && (comp_ratio < hi_comp_ratio)
        {
            quality_lo_lmt = quality_mid_lmt;
            lo_comp_ratio = mid_comp_ratio;
            quality_mid_lmt = (quality_hi_lmt + quality_lo_lmt) / 2;

            jpeg_DataBuffer = Encode(jpeg_SmallRawBuffer, &len, ret_quality, (int)n_ImageHeight, (int)n_ImageWidth, (int)n_ImageBpp, (int)quality_mid_lmt);
            mid_comp_ratio = (float)len / (float)small_img_size;
            free(jpeg_DataBuffer);
        }
    }

    if ((quality_lo_lmt == quality_mid_lmt) || (quality_hi_lmt == quality_mid_lmt))
    {
        quality = quality_mid_lmt;
        flag = 1;
    }

    iterations++;

    // the time issue
    time = end_time;

    time_span = difftime(end_time, start_time);
    sec_elapsed = (long)time_span;

    if (sec_elapsed > max_time)
    {
        quality = quality_mid_lmt;
        flag = 1;
    }
} // end of the while loop.

```

```

n_ImageHeight = n_store_height;
n_ImageWidth = n_store_width;

jpeg_DataBuffer = Encode(jpeg_RGB_Buffer, length, ret_quality, (int)n_ImageHeight, (int)n_ImageWidth, (int)n_ImageBpp, (int)quality);
// careful about the "length"... and NOT "len" here.
mid_comp_ratio = float(len) / small_img_size;

free(jpeg_SmallRawBuffer);

*ret_quality = quality;

free(jpeg_RGB_Buffer);

return jpeg_DataBuffer;

} // end of the overloaded Encode.

```

/\* Residual Converter \*/

```

BYTE *JPEG::Resolution_Convert(BYTE *jpeg_get_Buffer, int *bpp, int *res, int Height, int *Width)
{
    // res can be either 1 or 3.

    int have_bpp;
    int want_res;
    int mod_val, length, length2, length4, i, diff_len, diff, max, min;

    int val1, val2;
    int n_Height, n_Width;
    int mod_val, wid_diff_len, start_i;

    div_t mod_val;

    BYTE *jpeg_RGB_Buffer;

    have_bpp=*bpp;
    want_res=*res;

    n_Height=Height;
    n_Width=*Width;

    if (want_res<1)
        want_res=1;

    *res=want_res;

    mod_val=div(want_res-1,4);
    want_res=mod_val.rem+1;

    switch (want_res)
    {
        case (JPEG::ONE_BYTE):{
            if (have_bpp==3)
                length=n_Height*n_Width;
            else
                length=n_Height*n_Width*have_bpp;

            jpeg_RGB_Buffer=(BYTE *) malloc(sizeof(BYTE)*length);
            if (jpeg_RGB_Buffer==NULL){
                fprintf(stderr, "Memory alloc error\n");
                exit(1);
            }
            if (have_bpp==3)

                for(i=0;i<length;i++)
                {
                    val=0;
                    val+=jpeg_get_Buffer[3*i];
                    val+=jpeg_get_Buffer[(3*i)+1];
                    val+=jpeg_get_Buffer[(3*i)+2];

                    jpeg_RGB_Buffer[i]=(BYTE) (val/3);

                }

            else
                for(i=0;i<length;i++)
                    jpeg_RGB_Buffer[i]=jpeg_get_Buffer[i];

            *bpp=1;
            *Width=n_Width;

        }
        break;
    }
}

```

```

case JFIF::TWO_BYTE):{
    len=n_Height*n_Width*have_bpp;
    length=length/2;
    length2=length+(length-(2*length2));
    jpeg_RGB_Buffer=(BYTE *) malloc(sizeof(BYTE)*len);
    if (jpeg_RGB_Buffer==NULL){
        fprintf(stderr,"Memory alloc error\n");
        exit(1);
    }

    max=0;
    min=255*256;
    for (i=0;i<length2;i++)
    {
        val1=(int)jpeg_get_Buffer[i*2];
        val2=(int)jpeg_get_Buffer[(i*2)+1];

        val=(val1*256)+val2;
        if (val>=max)
            max=val;
        if (val<=min)
            min=val;

        //for the last guy
        if (length!=(2*length2))
        {
            val=(int)jpeg_get_Buffer[2*length2];

            if (val>=max)
                max=val;
            if (val<=min)
                min=val;

            diff=max-min;

            for (i=0;i<length2;i++)
            {
                val1=(int)jpeg_get_Buffer[i*2];
                val2=(int)jpeg_get_Buffer[(i*2)+1];

                val=(val1*256)+val2;

                jpeg_RGB_Buffer[i]=(BYTE)((val-min)*255/diff);
            }

            if (length!=(2*length2))
            {
                val=(int)jpeg_get_Buffer[2*length2];

                jpeg_RGB_Buffer[length2]=(BYTE)((val-min)*255/diff);
            }

            bpp=1;

        }

        break;
    }

case JFIF::THREE_BYTE):{
    len=n_Height*n_Width*have_bpp;
    length=length;
    width=len/(n_Height*3);
    for ((width*3*n_Height)!=len)
        width++;

```

```

        diff_len=(wid*3*n_Height)-len;
        len=len+wid_diff_len;

        jpeg_RGB_Buffer=(BYTE *) malloc(sizeof(BYTE)*(len));
        if (jpeg_RGB_Buffer==NULL) {
            fprintf(stderr,"Memory alloc error\n");
            exit(1);

        }

        for (i=0;i<length;i++)
            jpeg_RGB_Buffer[i]=jpeg_get_Buffer[i];

        for (i=length;i<len;i++)
            jpeg_RGB_Buffer[i]=0;

        type=3;
        width=wid;

    }

    break;

```

```

case JPEGR::FOUR_BYTE):{
    width=n_Height*n_Width*have_bpp;
    length=length/4;
    length4=length*4;
    if (length!=(4*length4))
        len=len+3;

    wid=len/(n_Height*3);

    if ((wid*3*n_Height)!=len)
        wid++;

    wid_diff_len=(wid*3*n_Height)-len;
    len=len+wid_diff_len;

    jpeg_RGB_Buffer=(BYTE *) malloc(sizeof(BYTE)*len);
    if (jpeg_RGB_Buffer==NULL) {
        fprintf(stderr,"Memory alloc error\n");
        exit(1);

    }

    for (i=0;i<length4;i++)
        jpeg_RGB_Buffer[i*3]=jpeg_get_Buffer[4*i];
        jpeg_RGB_Buffer[(i*3)+1]=jpeg_get_Buffer[(4*i)+1];
        jpeg_RGB_Buffer[(i*3)+2]=jpeg_get_Buffer[(4*i)+2];

    diff_len=length-(4*length4);

    for (i=0;i<diff_len;i++)
        jpeg_RGB_Buffer[(length4*3)+i]=jpeg_get_Buffer[(length4*4)+i];

    start_i=(3*length4)+diff_len;

    for (i=start_i;i<len;i++)
        jpeg_RGB_Buffer[i]=0;

    type=3;
    width=wid;

}

break;

default:
    break;

```



```

/*****
*
* Copyright (C) 2000, Louisiana State University, School of Medicine
*
* This DICOM object library was developed based on University of California,
* Davis UCDM DICOM Network Transport Libraries, in full compliance
* with the copyright note below. This version however contains conceptual
* deviations from the UCDM library, as well as important bug and performance
* fixes, and cannot be used/copied/distributed without our permission
*
* Technical Contact: oleg@bit.csc.lsu.edu
*
*****/

```

```

/*****
* PDU Service Classes:
* A-ASSOCIATE-RQ Class.
*
* Base Classes:
* ApplicationContext
* AbstractSyntax
* TransferSyntax
* PresentationContext
* MaximumStringLength
* ImplementationClass
* ImplementationVersion
* UserInformation
*
*****/

```

```

#ifdef _AARQ_HPP_INCLUDED_
#define _AARQ_HPP_INCLUDED_

class ApplicationContext
{
private:
    BYTE      ItemType;    // 0x10
    BYTE      Reserved1;   // 0x00
    UINT16    Length;

public:
    UID      ApplicationContextName;

    ApplicationContext();
    ApplicationContext(UID &);
    ApplicationContext(BYTE *);
    ~ApplicationContext();
    void      Set(UID &);
    void      Set(BYTE *);
    BOOL      Write(Buffer &);
    BOOL      Read(Buffer &);
    BOOL      ReadDynamic(Buffer &);
    UINT32    Size();
};

```

```

class AbstractSyntax
{
private:
    BYTE      ItemType;    // 0x30
    BYTE      Reserved1;   // 0x00
    UINT16    Length;

public:
    UID      AbstractSyntaxName;

    AbstractSyntax();
    AbstractSyntax(BYTE *);
    AbstractSyntax(UID &);
    ~AbstractSyntax();
    void      Set(UID &);
    void      Set(BYTE *);
    BOOL      Write(Buffer &);
    BOOL      Read(Buffer &);
    BOOL      ReadDynamic(Buffer &);
    UINT32    Size();
};

```

```

class TransferSyntax
{
private:
    BYTE      ItemType;    // 0x40

```

```

    BYTE    Reserved1; // 0x00
    UINT16   Length;

public:
    UINT    EndianType; // not really used so far
    UID     TransferSyntaxName;

    TransferSyntax();
    TransferSyntax(BYTE *);
    TransferSyntax(UID &);
    ~TransferSyntax();
    void     Set(UID &);
    void     Set(BYTE *);
    void     SetType(UINT    T) { EndianType = T; };
    BOOL     Write(Buffer &);
    BOOL     Read(Buffer &);
    BOOL     ReadDynamic(Buffer &);
    UINT32   Size();
};

```

```

class ImplementationClass
{
private:
    BYTE    ItemType; // 0x52
    BYTE    Reserved1; // 0x00
    UINT16   Length;

public:
    UID     ImplementationName;

    ImplementationClass();
    ImplementationClass(BYTE *);
    ImplementationClass(UID &);
    ~ImplementationClass();
    void     Set(UID &);
    void     Set(BYTE *);
    BOOL     Write(Buffer &);
    BOOL     Read(Buffer &);
    BOOL     ReadDynamic(Buffer &);
    UINT32   Size();
};

```

```

class ImplementationVersion
{
private:
    BYTE    ItemType; // 0x55
    BYTE    Reserved1; // 0x00
    UINT16   Length;

public:
    UID     Version;

    ImplementationVersion();
    ImplementationVersion(BYTE *);
    ImplementationVersion(UID &);
    ~ImplementationVersion();
    void     Set(UID &);
    void     Set(BYTE *);
    BOOL     Write(Buffer &);
    BOOL     Read(Buffer &);
    BOOL     ReadDynamic(Buffer &);
    UINT32   Size();
};

```

```

class SCPSCURoleSelect
{
private:
    BYTE    ItemType; // 0x54
    BYTE    Reserved1; // 0x00
    UINT16   Length;

public:
    BYTE    SCURole;
    BYTE    SCPRole;
    UID     SOPuid;

    SCPSCURoleSelect();
    ~SCPSCURoleSelect();
    BOOL     Write(Buffer &);
    BOOL     Read(Buffer &);
    BOOL     ReadDynamic(Buffer &);
    UINT32   Size();
};

```



);

class PresentationContext

```
{
    private:
        BYTE ItemType; // 0x20
        BYTE Reserved1; // 0x00
        BYTE Reserved2; // 0x00
        BYTE Reserved3; // 0x00
        BYTE Reserved4; // 0x00
        UINT16 Length;

    public:
        BYTE PresentationContextID;
        AbstractSyntax AbsSyntax;
        Array<TransferSyntax> TrnSyntax;

        PresentationContext();
        PresentationContext(AbstractSyntax &, TransferSyntax &);
        ~PresentationContext();
        void SetAbstractSyntax(AbstractSyntax &);
        void AddTransferSyntax(TransferSyntax &);
        BOOL Write(Buffer &);
        BOOL Read(Buffer &);
        BOOL ReadDynamic(Buffer &);
        UINT32 Size();
};
```

class MaximumSubLength

```
{
    private:
        BYTE ItemType; // 0x51
        BYTE Reserved1; // 0x00
        UINT16 Length; // 0x04
        UINT32 MaximumLength;

    public:
        MaximumSubLength();
        MaximumSubLength(UINT32);
        ~MaximumSubLength();
        void Set(UINT32);
        void Get();
        BOOL Write(Buffer &);
        BOOL Read(Buffer &);
        BOOL ReadDynamic(Buffer &);
        UINT32 Size();
};
```

class ExtendedNegotiation

```
{
    private:
        BYTE ItemType; // 0x56
        BYTE Reserved1; // 0x00
        BYTE RelationalDB;
        UINT16 Length;

    public:
        UID SOPuid;

        ExtendedNegotiation();
        ~ExtendedNegotiation();
        BOOL Write(Buffer &);
        BOOL Read(Buffer &);
        BOOL ReadDynamic(Buffer &);
        UINT32 Size();
};
```

class UserInformation

```
{
    private:
        BYTE ItemType; // 0x50
        BYTE Reserved1;
        UINT16 Length;

    public:
        UINT32 UserInfoBaggage;
        MaximumSubLength MaxSubLength;
        ImplementationClass ImpClass;
        ImplementationVersion ImpVersion;
        SCPSCURoleSelect SCPSCURole;
        ExtendedNegotiation ExtNegotiation;
};
```

```

    UseInformation();
    ~UseInformation();
    void SetMax(MaximumSubLength &);
    UINT16 GetMax();
    BOOL Write(Buffer &);
    BOOL Read(Buffer &);
    BOOL ReadDynamic(Buffer &);
    UINT16 Size();
};

class AAssociateRQ
{
private:
    BYTE ItemType; // 0x01
    BYTE Reserved1;
    UINT16 Length;
    UINT16 ProtocolVersion; // 0x01
    UINT16 Reserved2;
public:
    BYTE CalledApTitle[17]; // 16 bytes transfered
    BYTE CallingApTitle[17]; // 16 bytes transfered
    BYTE Reserved3[32];

    ApplicationContext ApplicationContext;
    Array<PresentationContext> PresContexts;
    UserInformation UserInfo;
public:
    AAssociateRQ();
    AAssociateRQ(BYTE *, BYTE *);
    virtual ~AAssociateRQ();
    void SetCalledApTitle(BYTE *);
    void SetCallingApTitle(BYTE *);
    void SetApplicationContext(ApplicationContext &);
    void SetApplicationContext(UID &);
    void AddPresentationContext(PresentationContext &);
    void ClearPresentationContexts();
    void SetUserInformation(UserInformation &);
    BOOL Write(Buffer &);
    BOOL Read(Buffer &);
    BOOL ReadDynamic(Buffer &);
    UINT32 Size();
};
#endif

```

```

/*****
*
* Copyright (C) 2000, Louisiana State University, School of Medicine
*
* This DLL object library was developed based on University of California,
* Davis UCDMC DICOM Network Transport Libraries, in full compliance
* with the copyright note below. This version however contains conceptual
* deviations from the UCDMC library, as well as important bug and performance
* fixes, and cannot be used/copied/distributed without our permission
*
* Technical Contact: oleg@bit.csc.lsu.edu
*
*****/

```

```

template <class DATATYPE>
class DataLink
{
public:
    DATATYPE Data;
    DataLink<DATATYPE> *prev, *next;

    DataLink() { prev = NULL; next = NULL; };
};

```

```

template <class DATATYPE>
class Array
{
private:
    unsigned int LastAccessNumber;
    unsigned int ArraySize;
    DataLink<DATATYPE> *first;
    DataLink<DATATYPE> *last;
    DataLink<DATATYPE> *LastAccess;

public:
    UINT ClearType;

    void RemoveAll()
    {
        if(ClearType == 1) { while(ArraySize) RemoveAt(0); }
    }

    void RemoveLast()
    {
        if(ArraySize<1) return;
        RemoveAt(ArraySize-1);
    }

    void SetSize(int new_size);
    void Swap(UINT index1, UINT index2);
    void Include(Array<DATATYPE>& a);
    bool IsEmpty() { return (ArraySize<=0); };
    BOOL RemoveAt(unsigned int);
    BOOL ClearArray ()
    {
        first = last = LastAccess = NULL;
        LastAccessNumber = 0;
        ArraySize = 0;
        return ( TRUE );
    }

    inline unsigned int
    GetSize() { return ArraySize ; };

    inline int
    GetUpperBound()
    {
        if (ArraySize<=0) return -1;
        else return ArraySize-1;
    }

    DATATYPE& Add(DATATYPE &);
    DATATYPE& Add();
    DATATYPE& Get(unsigned int);
    inline DATATYPE &
    operator [] (unsigned int Index)
    {
        return(Get(Index));
    }
};

```

```

virtual ~Array()
{
    RemoveAll();
}

void operator = (Array<DATATYPE> &array)
{
    RemoveAll();
    first = array.first; last = array.last;
    ArraySize = array.ArraySize; ClearType = FALSE;
}

// Constructors
Array()
{
    ArraySize = 0; first = NULL; last = NULL;
    LastAccess = NULL; LastAccessNumber = 0;
    ClearType = 1;
}

Array (UINT CT)
{
    ArraySize = 0; first = NULL; last = NULL;
    LastAccess = NULL; LastAccessNumber = 0;
    ClearType = CT;
}

Array (DATATYPE & d)
{
    ArraySize = 0; first = NULL; last = NULL;
    LastAccess = NULL; LastAccessNumber = 0;
    ClearType = 1;
    Add(d);
}

private:
void Move(DataLink<DATATYPE> *dmove, DataLink<DATATYPE> *dloc,
          bool insert_before_dloc);
void Swap(DataLink<DATATYPE> *d1, DataLink<DATATYPE> *d2);
inline DataLink<DATATYPE>*
GetDataLink(unsigned int Index);
} // end of class prototype
*****
* Element manipulation
*****
template class DATATYPE>
DATATYPE & Array<DATATYPE> :: Add(DATATYPE &Value)
{
    // record current end-of-chain element
    DataLink<DATATYPE> *dl = last;

    // chain on new element at tail of chain
    last = new DataLink<DATATYPE>;
    last->Data = Value;

    // set new element's backward pointer to point to former
    // end-of-chain element
    last->prev = dl;

    // set former end-of-chain's next pointer to point to new element
    if(dl) dl->next = last;
    else
    {
        // there was previously no "last" element so the one just
        // allocated must be the first
        first = last;
    }

    ++ArraySize;
    return Value;
}

template class DATATYPE>
DATATYPE & Array<DATATYPE> :: Add()
{
    // record current end-of-chain element
    DataLink<DATATYPE> *dl = last;

    // chain on new element at tail of chain
    last = new DataLink<DATATYPE>;

```

```

// set element's back pointer to point to former
// end of chain element
last->prev = dl;

// set former end-of-chain's next pointer to point to new element
if(dl->next == last)
else
{
    // there was previously no "last" element so the one just
    // allocated must be the first
    first = last;
}

++ArraySize;
return (last->Data);
}

template <class DATATYPE>
DATATYPE * Array<DATATYPE> :: Get(unsigned int Index)
{
    return (GetDataLink(Index)->Data); // will throw exception on NULL link
}

template <class DATATYPE>
DataLink<DATATYPE> * Array<DATATYPE> :: GetDataLink(unsigned int Index)
{
    if ( Index >= ArraySize ) return NULL;

    if ( LastAccess ) // Sort of access cache
    {
        int d = (int)LastAccessNumber-Index;
        if(d==0) // same as before
        {
            return LastAccess;
        }
        else if(d == -1) // next after the most recently accessed
        {
            LastAccess = LastAccess->next;
            ++LastAccessNumber;
            return LastAccess;
        }
        else if(d == 1) // previous before the most recently accessed
        {
            LastAccess = LastAccess->prev;
            --LastAccessNumber;
            return LastAccess;
        }
    }

    // locate requested element by following pointer chain
    // decide which is faster -- scan from head or scan from tail
    DataLink<DATATYPE> *dl;
    unsigned int rIndex = Index;

    if(Index < ArraySize / 2)
    {
        // requested element closer to head -- scan forward
        dl = first;
        ++Index;
        while(--Index > 0) dl = dl->next;
    }
    else
    {
        // requested element closer to tail -- scan backwards
        dl = last;
        Index = (ArraySize - Index);
        while(--Index > 0) dl = dl->prev;
    }
    LastAccess = dl;
    LastAccessNumber = rIndex;
    return (dl);
}

template <class DATATYPE>
BOOL Array<DATATYPE> :: RemoveAt(unsigned int Index)
{
    DataLink<DATATYPE> *dl = GetDataLink(Index);

```

```

    if(!dl) return FALSE;

    // Relink chain around element to be deleted
    if(dl->prev) dl->prev->next = dl->next;
    else first = dl->next;

    if(dl->next) dl->next->prev = dl->prev;
    else last = dl->prev;

    delete dl;
    --ArraySize;
    LastAccess = NULL;
    LastAccessNumber = 0;
    return TRUE;
}
/*****
*
* Include all elements from Array "a" into this array.
* Note: Array "a" becomes empty after inclusion !
*
*****/
template <class DATATYPE>
void Array<DATATYPE>:: Include(Array<DATATYPE>& a)
{
    if(!ClearType || !a.ClearType) return; // Cannot include linked arrays
    if(a.ArraySize<=0) return; // Nothing to include
    if(ArraySize>0)
    {
        last->next=a.first; a.first->prev = last;
        last = a.last;
    }
    else
    {
        first = a.first; last = a.last;
    }
    ArraySize += a.ArraySize;
    LastAccess = NULL;
    LastAccessNumber = 0;
    a.ClearType = 0;
    a.ClearArray();
}
/*****
*
* Move one linked element before or after the other
*
*****/
template <class DATATYPE>
void Array<DATATYPE>:: Move(DataLink<DATATYPE> *dmove,
    DataLink<DATATYPE> *dloc, bool insert_before_dloc)
{
    if(!dmove || !dloc || dmove==dloc) return;

    // Do we need to move anything at all?
    if(insert_before_dloc)
    {
        if(dmove->next==dloc) return; // already there
    }
    else insert after dloc
    {
        if(dmove->prev==dloc) return; // already there
    }

    // Remove dmove from its present location
    if(dmove->prev) dmove->prev->next = dmove->next;
    else first = dmove->next;

    if(dmove->next) dmove->next->prev = dmove->prev;
    else last = dmove->prev;

    // Insert dmove into its new location
    if(insert_before_dloc)
    {
        dmove->prev = dloc->prev;
        dmove->next = dloc;
        if(dloc->prev) dloc->prev->next = dmove;
        else first = dmove;
        dloc->prev = dmove;
    }
    else move after dloc
    {

```

```

        dmove->next = dloc->next;
        dmove->prev = dloc;
        if (dmove->next) dloc->next->prev = dmove;
        else last = dmove;
        dloc->next = dmove;
    }
    LastAccessNumber = 0;
    LastAccess = NULL;
}

/*****
 *
 *   Swapping two elements
 *
 *****/
template <class DATATYPE>
void Array<DATATYPE> :: Swap(DataLink<DATATYPE> *d1,
                             DataLink<DATATYPE> *d2)
{
    if(!d1 || !d2 || d1==d2) return;
    DataLink<DATATYPE> *dt = d1->next;
    if(dt)
    {
        if(d1==d2) // neighbors
        {
            if(d2->next) Move(d1, d2->next, true);
            else if(d1->prev) Move(d2, d1->prev, false);
            else // array contains only d1 and d2
            {
                first=d2; last=d1;
                d2->next = d1; d2->prev = NULL;
                d1->prev = d2; d1->next = NULL;
            }
        }
        else // not neighbors
        {
            Move(d1,d2,true); Move(d2,dt,true);
        }
    }
    else
    {
        dt = d2->next; if(!dt) return; // impossible
        if(d1==d2) // neighbors
        {
            if(d1->next) Move(d2, d1->next, true);
            else if(d2->prev) Move(d1, d2->prev, false);
            else // array contains only d1 and d2
            {
                first=d1; last=d2;
                d1->next = d2; d1->prev = NULL;
                d2->prev = d1; d2->next = NULL;
            }
        }
        else // not neighbors
        {
            Move(d2,d1,true); Move(d1,dt,true);
        }
    }
    LastAccessNumber = 0;
    LastAccess = NULL;
}

template <class DATATYPE>
void Array<DATATYPE> :: Swap(UINT index1, UINT index2)
{
    Swap(GetDataLink(index1), GetDataLink(index2));
}

/*****
 *
 *   Setting to a certain size
 *
 *****/
template <class DATATYPE>
void Array<DATATYPE> :: SetSize(int new_size)
{
    if(new_size<0) return;
    if(new_size == 0) { RemoveAll(); return; };
    int n;
    int d = ArraySize - new_size;
    if(d>0) Remove last elements

```





```

// basicTypes.h: interface for the basic DICOM type classes.
//
/////////////////////////////////////////////////////////////////

#ifndef BASICTYPES_H_INCLUDED_
#define BASICTYPES_H_INCLUDED_

#include "Basic.hpp"
#include <fstream>

// 2-Dimensional point
class Point2D
{
public:
    double x,y;

    Point2D() { x=0.0; y=0.0; };
    Point2D(double a, double b) { x=a; y=b; };
    Point2D(Point2D& p) { x=p.x; y=p.y; };

    bool SerializePoint2D(FILE* fp, bool is_loading)
    {
        return (::SerializeDouble(fp,x,is_loading) &&
            ::SerializeDouble(fp,y,is_loading));
    }
};

// Callback object
class CallbackObject
{
private:
    void* m_ptrArgument;
    int (*m_ptrFunction)(void* arg, UINT param1=0, UINT param2=0);
public:
    static const BYTE m_CBConnectingToAE;
    static const BYTE m_CBConnectionFailed;
    static const BYTE m_CBSendingRequest;
    static const BYTE m_CBGettingResponse;
    static const BYTE m_CBResponseReceived;
    static const BYTE m_CBCancelSent;
    static const BYTE m_CBIsCancelled;
    static const BYTE m_CBDQRTaskSchedule;

    bool SetCallback(int (*func)(void* a, UINT param1=0, UINT param2=0),
        void* arg=NULL)
    {
        m_ptrFunction = func;
        m_ptrArgument = arg;
        return (m_ptrFunction != NULL);
    };
    int InvokeCallback(UINT param1=0, UINT param2=0)
    {
        if(m_ptrFunction)
        {
            try
            {
                return m_ptrFunction(m_ptrArgument, param1, param2);
            }
            catch(...) { return 0; }
        }
        else return 0;
    };
    // constructors
    CallbackObject() { m_ptrArgument=NULL; m_ptrFunction=NULL; };
    CallbackObject(int (*func)(void* a, UINT param1=0, UINT param2=0),
        void* arg=NULL)
    {
        SetCallback(func,arg);
    };
    CallbackObject(CallbackObject& cbo)
    {
        m_ptrArgument = cbo.m_ptrArgument;
        m_ptrFunction = cbo.m_ptrFunction;
    };
};

// UID
class UID

```

```

{
private:
    BYTE    uid[65];
    UINT    Length;
public:
    void    ClearUID() { ZeroMem(uid, 64); Length = 0; };
    void    Set(BYTE *s)
    {
        if(!) return;
        ZeroMem(uid, 64);
        strcpy((char *) uid, (char *) s);
        Length = strlen((char*) uid);
    };
    void    Set(UID &u) { (*this) = u; };
    void    Set(char *s) { this->Set((BYTE *) s); };
    void    SetLength(UINT L)
    {
        Length = L;
        while (L < 65) uid[L++] = '\0';
    };
    BOOL    operator == (UID &ud)
    {
        if(!strcmp((char*) GetBuffer(), (char*) ud.GetBuffer()))
            return(TRUE);
        return(FALSE);
    };
    BOOL    operator != (UID &ud) { return (!(*this)==ud); };
    BOOL    operator = (UID &ud)
    {
        ByteCopy(uid, ud.GetBuffer(), 64);
        SetLength(ud.GetSize());
        return(TRUE);
    };
    BYTE    *GetBuffer() { return(&uid[0]); };
    UINT    GetSize() { return ( Length ); };

    UID()          { ClearUID(); };
    UID(BYTE *s)   { Set(s); };
    UID(char *s)   { Set((BYTE*)s); };
};

// DateTime
class DateTime
{
public:
    void SetNumericTime(double t);
    double GetNumericTime();
    void SetNumericDate(int dt);
    int GetNumericDate();
    static const BYTE    DateFormat;
    static const BYTE    TimeFormat;
    static const BYTE    DateTimeFormat;
    static const BYTE    UnknownFormat;

    void    SetCurrentDateTime();
    void    SetCurrentTime();
    void    SetCurrentDate();
    inline void    ClearDate() { m_Year=-1; m_Month=0; m_Day=0; };
    inline void    ClearTime() { m_Hour=-1; m_Minute=0; m_Second=0.0; };
    inline void    ClearDateTime() { ClearDate(); ClearTime(); };
    inline bool    EmptyDate() { return (m_Year<0); };
    inline bool    EmptyTime() { return (m_Hour<0); };
    inline bool    EmptyDateTime() { return EmptyDate()||EmptyTime(); };
    bool    SerializeDateTime(FILE* fp, bool is_loading);
    bool    SetTime(char* str);
    bool    SetDate(char *str);
    bool    SetDateTime(int y, int mo=0, int d=0, int h=0,
                        int mi=0, double s=0);
    bool    SetTime(int h, int m=0, double s=0);
    bool    SetDate(int y, int m=0, int d=0);
    bool    SetSecond(double s);
    bool    SetMinute(int m);
    bool    SetHour(int h);
    bool    SetDay(int d);
    bool    SetMonth(int m);
    bool    SetYear(int y);
    bool    FormatDateTime(char *str, int max_len, bool dicom_format);
    bool    FormatDate(char *str, int max_len, bool dicom_format);
    bool    FormatTime(char *str, int max_len, bool dicom_format);
};

```

```

int      SetDateTime(char* str, BYTE format=UnknownFormat);
int      GetYear()      { return m_Year; };
int      GetMonth()     { return m_Month; };
int      GetDay()       { return m_Day; };
int      GetHour()      { return m_Hour; };
int      GetMinute()    { return m_Minute; };
double   GetSecond()    { return m_Second; };
struct tm GetTM();
DateTime GetLower();
DateTime GetUpper();

bool      operator == (DateTime& d);
bool      operator != (DateTime& d);
bool      operator > (DateTime& d);
bool      operator >= (DateTime& d);
bool      operator < (DateTime& d);
bool      operator <= (DateTime& d);

DateTime() ;
DateTime(DateTime& d);
virtual ~DateTime();

private:
int      m_Year, m_Month, m_Day, m_Hour, m_Minute;
double   m_Second;

bool      Format(char *dest, int max_len, const char *format);
};

// DateTimeSegment
class DateTimeSegment
{
public:
void      SetNumericTime(double t);
void      SetNumericDate(int d);
void      Normalize();
inline void SetStart(DateTime& d) { m_Start=d; };
inline void SetEnd(DateTime& d) { m_End=d; };
inline void SetDateTimeSegment(DateTime& start, DateTime& end)
        { m_Start=start; m_End=end; };
inline void ClearDateTimeSegment()
        { m_Start.ClearDateTime(); m_End.ClearDateTime(); };
bool      SetDateTimeSegment(char* str, BYTE format=DateTime::UnknownFormat);
bool      Intersects(DateTimeSegment& d);
bool      ContainsDateTime(DateTime& dt);
bool      SerializeDateTimeSegment(FILE *fp, bool is_loading);
bool      FormatDateTime(char *str, int max_len, bool dicom_format);
bool      FormatTime(char *str, int max_len, bool dicom_format);
bool      FormatDate(char *str, int max_len, bool dicom_format);
bool      EmptyDateTimeSegment()
        { return (m_Start.EmptyDateTime() && m_End.EmptyDateTime()); };
static char*
        FormatStaticDateTimeString(char* strdest, int max_dest_len,
        bool dest_format, char* strsource, BYTE source_format);
DateTime  GetStart()      { return m_Start; }
DateTime  GetEnd()        { return m_End; }
DateTimeSegment Expand();

DateTimeSegment();
DateTimeSegment(DateTimeSegment& dts);
virtual ~DateTimeSegment();

private:
DateTime  m_Start, m_End;
};

// ApplicationEntity
class ApplicationEntity
{
public:
bool      ae_Served;
bool      ae_UseMoveAsGet;
char      ae_Title[20], ae_partnerTitle[20];
char      ae_Location[32];
char      ae_Comments[64];
int      ae_Port, ae_PortServer;
int      ae_Timeout;
BYTE      ae_IP1, ae_IP2, ae_IP3, ae_IP4;

```

```

bool    SetComments(char* s);
bool    SetLocation(char* s);
bool    SetPartnerTitle(char* s);
bool    SetTitle(char* s);
bool    SetApplicationEntity(char* title,
                             BYTE ip1=127, BYTE ip2=0, BYTE ip3=0, BYTE ip4=1,
                             int port=104, int port_ser=104, int timeout=300,
                             char* location="Unspecified location",
                             char* comments="No comments",
                             bool useMoveAsGet=false);

bool    SerializeAE(FILE* fp, bool is_loading);
bool    operator == (ApplicationEntity& a)
{ return (strcmp(ae_Title, a.ae_Title)==0); };

ApplicationEntity();
ApplicationEntity(ApplicationEntity& a);
ApplicationEntity(char* title,
                  BYTE ip1=127, BYTE ip2=0, BYTE ip3=0, BYTE ip4=1,
                  int port=104, int port_ser=104, int timeout=300,
                  char* location="Unspecified location",
                  char* comments="No comments",
                  bool useMoveAsGet=false);

virtual ~ApplicationEntity();

char*    GetPortString();
char*    GetPortServerString();
char*    GetIPString();

private:
    char    a_IP_str[20], ae_Port_str[8], ae_PortServer_str[8];
};

// ApplicationEntityList
class ApplicationEntityList : public Array<ApplicationEntity>
{
public:
    void    SetServedStatus(int port, bool status);
    bool    GetAELocation(UINT aeindex, char* loc);
    bool    SetLocalAE(BYTE ip1, BYTE ip2, BYTE ip3, BYTE ip4,
                      char* title=NULL);
    bool    IdentifyAE(char* title, ApplicationEntity& aeFound);
    bool    SerializeAEList(bool is_loading, char* filename=NULL);
    bool    SetCurrentIndex(int n);
    int     IdentifyAEIndex(char* title);
    UINT    GetCurrentIndex() { return m_CurrentIndex; };
    UINT    GetLocalIndex() { return m_LocalIndex; };
    ApplicationEntity&
        GetLocalAE() { return Get(m_LocalIndex); };
    ApplicationEntity&
        GetCurrentAE()
        {
            Get(m_CurrentIndex).SetPartnerTitle(GetLocalAE().ae_Title);
            return Get(m_CurrentIndex);
        };
    ApplicationEntityList();
    virtual ~ApplicationEntityList();

private:
    char    m_SerFile[MAX_PATH];
    UINT    m_CurrentIndex;
    const UINT m_LocalIndex; // always 0

    void    LoadDefaults();
    bool    SetLocalAE(UINT n);
    bool    CreateLocalAE(BYTE ip1, BYTE ip2, BYTE ip3, BYTE ip4,
                        char* title=NULL);
};

#endif // !defined(_BASICTYPES_H_INCLUDED_)

```

```

/*****
*
* Copyright (C) 2000, Louisiana State University, School of Medicine
*
* This DICOM object library was developed based on University of California,
* Davis UCDM DICOM Network Transport Libraries, in full compliance
* with the copyright note below. This version however contains conceptual
* deviations from the UCDM library, as well as important bug and performance
* fixes, and cannot be used/copied/distributed without our permission
*
* Technical Contact: oleg@bit.csc.lsu.edu
*
*****/

```

```

*****/

```

```

class BufferSpace
{
public:
    BOOL        isTemp;
    BYTE        *Data;
    INT         BufferSize;
    UINT        Index;

    BufferSpace(UINT);
    BufferSpace();
    ~BufferSpace();
};

class Buffer
{
protected:
    UINT        BreakSize;
    UINT        InEndian;
    UINT        OutEndian;
    INT         InSize;
    INT         OutSize;
    Array<BufferSpace*> Incoming;
    Array<BufferSpace*> Outgoing;

    BOOL        ReadBlock();

public:
    BOOL        SetBreakSize(UINT);
    BOOL        SetIncomingEndian(UINT);
    BOOL        SetOutgoingEndian(UINT);
    BOOL        Flush();
    BOOL        Flush(UINT Bytes);
    BOOL        Kill(UINT);
    BOOL        Read(BYTE *, UINT);
    BOOL        Write(BYTE *, UINT);
    BOOL        Fill(UINT);
    inline UINT GetIncomingEndian() { return ( InEndian ); };
    inline UINT GetOutgoingEndian() { return ( OutEndian ); };
    inline UINT GetSize() { return ( InSize ); };

    Buffer & operator >> (BYTE &x);
    Buffer & operator >> (UINT16 &x);
    Buffer & operator >> (UINT32 &x);
    inline Buffer & operator >> (char &x)
    { return ( (*this)>>(BYTE &x) ); };
    inline Buffer & operator >> (INT16 &x)
    { return ( (*this)>>(UINT16 &x) ); };
    inline Buffer & operator >> (INT32 &x)
    { return ( (*this)>>(UINT32 &x) ); };

    Buffer & operator << (BYTE &x);
    Buffer & operator << (UINT16 &x);
    Buffer & operator << (UINT32 &x);
    inline Buffer & operator << (char &x)
    { return ( (*this)<<(BYTE &x) ); };
    inline Buffer & operator << (INT16 &x)
    { return ( (*this)<<(UINT16 &x) ); };
    inline Buffer & operator << (INT32 &x)
    { return ( (*this)<<(UINT32 &x) ); };

    virtual INT    ReadBinary(BYTE *, UINT)    = 0;
    virtual BOOL    SendBinary(BYTE *, UINT)    = 0;

    Buffer();
    virtual ~Buffer();
};

```

CONFIDENTIAL

```

/*****
*
* Copyright (C) 2000, Louisiana State University, School of Medicine
*
* This DICOM object library was developed based on University of California,
* Davis UCDM DICOM Network Transport Libraries, in full compliance
* with the copyright note below. This version however contains conceptual
* deviations from the UCDM library, as well as important bug and performance
* fixes, and cannot be used/copied/distributed without our permission
*
* Technical Contact: oleg@bit.csc.lsu.edu
*
*****/

```

```

#if !defined(_CCTYPES_H_INCLUDED_)
#define _CCTYPES_H_INCLUDED_

/* CC Types */

#ifdef SOLARIS
# define SYSTEM_V
#endif

#ifdef WIN32
typedef unsigned int BOOL;
#endif
typedef unsigned int UINT;
typedef unsigned short UINT16;
#ifdef _BASETSD_H_ // MSVC 6.0 define: typedef unsigned int UINT32 [gz]
typedef unsigned long UINT32;
#endif
typedef unsigned char UINT8;
#ifdef WIN32
typedef unsigned char BYTE;
#endif
typedef signed char INT8;
typedef signed short INT16;
#ifdef _BASETSD_H_ // MSVC 6.0 define: typedef int INT32 [gz]
typedef signed long INT32;
#endif
#ifdef WIN32
typedef signed int INT;
#endif

#ifdef TRUE
#define TRUE ((UINT) 1)
#endif
#ifdef FALSE
#define FALSE ((UINT) 0)
#endif

#ifdef LITTLE_ENDIAN
#undef LITTLE_ENDIAN
#endif
#ifdef BIG_ENDIAN
#undef BIG_ENDIAN
#endif

# define LITTLE_ENDIAN 1
# define BIG_ENDIAN 2
#ifdef NATIVE_ENDIAN
# define NATIVE_ENDIAN LITTLE_ENDIAN
#endif

#endif

```

```

////////////////////////////////////
// database.h: interface for DICOMRecord class.
//
////////////////////////////////////

#ifdef !defined( DICOM_DATABASE_H_INCLUDED_)
#define DICOM_DATABASE_H_INCLUDED_

#include "dicom.hpp"
#include "cctypes.h" // Added by ClassView

class DICOMRecord
{
public:
    const static BYTE LevelInvalid;
    const static BYTE LevelPatient;
    const static BYTE LevelStudy;
    const static BYTE LevelSeries;
    const static BYTE LevelImage;

    void SetAtRoot(BYTE root, DICOMRecord* pDR=NULL);
    void SetRecord(char *pID, char *pName, int pBDate,
        double pBTime, char* stIUnstID,
        char* stID, char* aNum, char* stImNum,
        int stDate, double stTime,
        char* serInstUID, char* mod, char* serNum,
        char* SOPUID, char* imNum, char* fName);
    void SetRecord(char *pID, char *pName, DateTimeSegment* pBDate,
        DateTimeSegment* pBTime, char* stIUnstID,
        char* stID, char* aNum, char* stImNum,
        DateTimeSegment* stDate, DateTimeSegment* stTime,
        char* serInstUID, char* mod, char* serNum,
        char* SOPUID, char* imNum, char* fName);

    void SetRetrieveAEs(char* aelist);
    void ClearAtCurrentLevel();
    void ClearDICOMRecord(bool remove_file=false);
    void RemoveOTModality();
    void ExpandDateTime();
    void WriteIntoDICOMObject(DICOMObject& dob, DICOMObject* dob_mask=NULL);
    bool HasUniquePrimaryKeys();
    bool WriteIntoDICOMQR(DICOMObject& dob, bool cfind);
    bool WriteIntoTextFile(char* filename);
    bool ChangeLevel(bool step_down, BYTE qr_root);
    bool AddRetrieveAE(char* aetitle);
    bool RemoveRetrieveAE(char* aetitle);
    bool FormatStudyTime(char *str, int max_len, bool dicom_format);
    bool FormatStudyDate(char *str, int max_len, bool dicom_format);
    bool FormatPatientBirthTime(char *str, int max_len, bool dicom_format);
    bool FormatPatientBirthDate(char *str, int max_len, bool dicom_format);
    bool SerializedDICOMRecord(FILE* fp, bool is_loading);
    bool SetRecordOnLevel(DICOMObject& dob);
    bool SetRecord(DICOMObject& DOB, char* filename);
    bool SetRecord(DICOMRecord& d);
    bool SetRecord(char* filename);
    bool MatchDICOMString(char* exact, char* mask,
        bool case_sensitive,
        bool first_level=false);

    bool operator==(DICOMRecord& dr);

    BYTE FindQLevel();
    BYTE GetQLevel() { return m_QLevel; };
    char* GetFileName() { return m_Filename; };
    char* GetRetrieveAEs() { return m_RetrieveAEs; };
    char* GetPatientID() { return m_PatientID; };
    char* GetPatientName() { return m_PatientName; };
    char* GetStudyInstUID() { return m_StudyInstUID; };
    char* GetStudyID() { return m_StudyID; };
    char* GetAccessionNumber() { return m_AccessionNumber; };
    char* GetStudyImagesNum() { return m_StudyImagesNum; };
    char* GetSeriesInstUID() { return m_SeriesInstUID; };
    char* GetModality() { return m_Modality; };
    char* GetSeriesNum() { return m_SeriesNum; };
    char* GetSOPInstUID() { return m_SOPInstUID; };
    char* GetImageNum() { return m_ImageNum; };
    char* GetRetrieveAE(char* aetitle, UINT aetitle_len, UINT n=1);
    int CompareAtLevel(DICOMRecord& dr, BYTE lev=LevelImage);
    UINT GetRetrieveAEsCount();
    DateTimeSegment GetPBirthTime() { return m_PBirthTime; };

```



```

DateTimeSegment GetPBirthTime() { return m_PBirthDate; };
DateTimeSegment GetStudyTime() { return m_StudyTime; };
DateTimeSegment GetStudyDate() { return m_StudyDate; };

DICOMRecord();
DICOMRecord(DICOMRecord& d);
virtual ~DICOMRecord();

private:
const static BYTE InsertUnique;
const static BYTE InsertNonEmpty;
const static BYTE InsertAny;

BYTE m_Level;
char m_Filename[MAX_PATH+1];
char m_RetrieveAEs[65];
char m_PatientID[65];
char m_PatientName[65];
char m_StudyInstUID[65];
char m_StudyID[17];
char m_AccessionNumber[17];
char m_StudyImagesNum[13];
char m_SeriesInstUID[65];
char m_Modality[3];
char m_SeriesNum[13];
char m_SOPInstUID[65];
char m_ImageNum[13];
DateTimeSegment
m_PBirthTime, m_PBirthDate, m_StudyTime, m_StudyDate;

void InsertStringForQLevel(DICOMObject& dob);
bool InsertString(DICOMObject& dob, UINT16 group, UINT16 element,
char *str, BOOL alloc, BYTE how=InsertAny);
}

////////////////////////////////////////////////////
// DICOMDatabase class.
////////////////////////////////////////////////////
class DICOMDatabase
{
public:
const static BYTE RetrieveRelational;
const static BYTE RetrieveHierarchical;
const static BYTE MatchRelational;
const static BYTE MatchHierarchical;
const static BYTE RecordFound;
const static BYTE FindMore;
const static BYTE RecordsEnd;
ApplicationEntityList db_AEList;

void EnableDisplayNew(bool b) { db_DisplayRecordsFlag=b; };
void DisplayRecords(bool from_local);
void* StartSearch(DICOMObject& dob, const BYTE how,
char* filename=NULL);
void* StartSearch(char* filename, const BYTE how);
virtual void StopSearch(void* pDR);
virtual void DisplayRecords(Array<DICOMRecord> &a, bool from_local);
bool DBAdd(char* filename, bool copy_into_db_directory=false);
bool AttachDirectory(char *directory,
bool include_subdirectories=true);
bool ImportDirectory(char* directory,
bool include_subdirectories=true);
bool UnImportDirectory(char *directory,
bool include_subdirectories=true);
bool UnAttachDirectory(char *directory,
bool include_subdirectories=true);
virtual bool GetFromLocal(DICOMDataObject& ddo_mask);
virtual bool SetRecordCount(int c);
virtual bool DBAdd(DICOMRecord& dr);
virtual bool DBAdd(DICOMObject* dob, PDU_Service* pdu);
virtual bool InitializeDataBase(char* directory,
void (*disp)(Array<DICOMRecord>&, bool));
virtual BYTE RetrieveNext(void* pDR, DICOMObject& dob_found);
virtual BYTE MatchNext(void* pDR, DICOMObject &dob_found,
DICOMObject* dob_mask=NULL);
char* GetMostRecentFile()
{ return db_MostRecentRecord.GetFileName(); };
int DBRemove(char* filename, bool use_filename=false);

```





```

////////////////////////////////////
// DICOMViewLog: interface for the DICOMView class.
//
////////////////////////////////////

#ifndef DICOMVIEW_H_INCLUDED_
#define DICOMVIEW_H_INCLUDED_

#include "dicomview.hpp"

// #if _MSC_VER > 1000
// #pragma once
// #endif // _MSC_VER > 1000

class DICOMView
{
public:
    virtual void Load(const char* pText)=0;
    void Load(VR* vr);
    virtual void Load(DICOMObject& DO);
    void ReportTime();
    void AttachRTC(RTC* rtc);
    virtual bool LoadFile(char* filename);
    virtual bool IgnoreVR(VR* vr) { return false; }
    DICOMView();
    virtual ~DICOMView();

protected:
    char m_SequenceMark;
    int m_SequenceLevel;
    int m_numElements;
    RTC* m_AttachedRTC;

    void MarkSequence(bool start);
};

////////////////////////////////////
// Interface for the DICOMViewLog class.
//
////////////////////////////////////

class DICOMViewLog : public DICOMView
{
public:
    void Load(const char* pText );
    void Load(DICOMObject& DO) { DICOMView::Load(DO); };
    bool CreateDVL(char* filename, RTC* rtc=NULL);
    DICOMViewLog();
    virtual ~DICOMViewLog();
protected:
    char* m_Filename;
    FILE* m_pFile;

    bool IgnoreVR(VR* vr);
    bool RefreshText(char* tbuffer, long max_length);
};

#endif // !defined(DICOMVIEW_H_INCLUDED_)

```

```

// dmodules.h: interface for the DModule class.
//
///////////////////////////////////////////////////////////////////

#ifndef AFX_DMODULES_H__3F70C2E3_CF5D_11D3_9760_00105A21774F__INCLUDED_
#define AFX_DMODULES_H__3F70C2E3_CF5D_11D3_9760_00105A21774F__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#include "dmodule.hpp"

class DModule
{
protected:
    virtual void      ClearAllParameters()=0;
    virtual bool      HasData()=0;
    virtual bool      ReadDO(DICOMObject &dob, char* fname=NULL)=0;
    virtual bool      WriteDO(DICOMObject &dob)=0;

    DModule() {}
    virtual ~DModule() {};
public:
    virtual bool      WriteFile(char* filename, PDU_Service* pPDU=NULL);
    virtual bool      ReadFile(char* filename, bool preview=false,
                               PDU_Service* pPDU=NULL);
};

class DModule_ImageSeries : DModule
{
public:
    void      GetPixelSpacing(double& dx, double& dy);
    bool      ReadDO(DICOMObject &dob, char* fname=NULL);
    bool      WriteDO(DICOMObject &dob);
    bool      ReadFile(char* filename, bool preview=false,
                       PDU_Service* pPDU=NULL);
    bool      WriteFile(char* filename);
protected:
    char      m_Filename[MAX_PATH+1];
    UINT16    m_SamplesPerPixel, m_Width, m_Height;
    UINT16    m_BitsAllocated, m_BitsStored, m_HighBit;
    int       m_NumberOfFrames;
    Point2D   m_PixelSpacing;
    DICOMObject m_DOB;
    PDU_Service m_PDU;

    void      ClearAllParameters();
    virtual bool ReadPixels(VR * vr, DICOMRecord& dr)=0;
    virtual bool WritePixels(VR * vr)=0;

    DModule_ImageSeries() { ClearAllParameters(); };
    virtual ~DModule_ImageSeries() {};
};

#endif // !defined(AFX_DMODULES_H__3F70C2E3_CF5D_11D3_9760_00105A21774F__INCLUDED_)

```





```

/*****
 * Copyright (C) 2000, Louisiana State University, School of Medicine
 *
 * This DICOM object library was developed from the University of California,
 * Davis UCDMC DICOM Network Transport Libraries, in full compliance
 * with the free access licence and copyright note below. The DICOM object
 * library however contains conceptual deviations from the UCDMC library,
 * as well as important bug and performance fixes, and CANNOT be
 * used/copied/distributed/modified without our permission.
 *
 * Technical Contact: oleg@bit.csc.lsu.edu
 *
 *****/
/*****
 * Sorted / Fixed, Fast-Access Array.
 *****/

```

```

template <class KEYTYPE, class DATATYPE>
class FixedArray
{
public:
    UINT      ArraySize;
    UINT      Top;
    KEYTYPE   *KeyTable;
    DATATYPE  *DataTable;

    FixedArray (UINT, BOOL);
    ~FixedArray ();
    BOOL      Sort();
    DATATYPE  & Add(KEYTYPE &, DATATYPE &);
    DATATYPE  & Add(DATATYPE &);
    INT       IndexOf(KEYTYPE &);
    DATATYPE  & Get(INT Index)
    { return(DataTable[Index]); };
    BOOL      RemoveAt(INT);
    DATATYPE  & operator [] (INT Index)
    { return(Get(Index)); };
    UINT      GetSize()
    { return ( Top ); };
    UINT      GetAllocationSize()
    { return ( ArraySize ); };
};

```

```

template <class KEYTYPE, class DATATYPE>
class FixedArrayElement
{
public:
    KEYTYPE   Key;
    DATATYPE  Data;
    UINT      operator > (FixedArrayElement &FAE)
    { return ( Key > FAE.Key ); };
    UINT      operator < (FixedArrayElement &FAE)
    { return ( Key < FAE.Key ); };
    UINT      operator == (FixedArrayElement &FAE)
    { return ( Key == FAE.Key ); };
};

```

```

template <class KEYTYPE, class DATATYPE>
FixedArray<KEYTYPE, DATATYPE> :: FixedArray (
    UINT      aSize,
    BOOL      UseKeys)
{
    ArraySize = aSize;
    Top = 0;
    if ( UseKeys )      KeyTable = new KEYTYPE [ aSize ];
    else                KeyTable = NULL;
    DataTable = new DATATYPE [ aSize ];
}

```

```

template <class KEYTYPE, class DATATYPE>
FixedArray<KEYTYPE, DATATYPE> :: ~FixedArray ()
{
    if ( KeyTable )
        delete KeyTable;
    if ( DataTable )
        delete DataTable;
}

```



```

template < class KEYTYPE, class DATATYPE>
BOOL FixArray<KEYTYPE, DATATYPE> :: Sort()
{
    INT Index;
    FixedArrayElement < KEYTYPE, DATATYPE > FAE;
    PQFAE FixedArrayElement<KEYTYPE, DATATYPE> > PQFAE;

    if ( ! KeyTable )
        return ( FALSE ); // Not a sortable array

    Index = 0;
    while ( Index < Top )
    {
        FAE.Key = KeyTable [ Index ];
        FAE.Data = DataTable [ Index ];
        PQFAE.Push(FAE);
        ++Index;
    }
    Index = 0;
    while ( Index < Top )
    {
        FAE = PQFAE.Pop ();
        KeyTable [ Index ] = FAE.Key;
        DataTable [ Index ] = FAE.Data;
        ++Index;
    }
    return ( TRUE );
}

```

```

template < class KEYTYPE, class DATATYPE>
DATATYPE & FixArray<KEYTYPE, DATATYPE> :: Add(
    KEYTYPE &Key,
    DATATYPE &Data)
{
    if (Top < ArraySize)
    {
        if ( KeyTable )
            KeyTable [ Top ] = Key;
        if ( DataTable )
            DataTable [ Top ] = Data;
        ++Top;
    }
    return ( DataTable [ Top-1 ] );
}

```

```

template < class KEYTYPE, class DATATYPE>
DATATYPE & FixArray<KEYTYPE, DATATYPE> :: Add(
    DATATYPE &Data)
{
    if ( Top < ArraySize )
    {
        if ( DataTable )
            DataTable [ Top ] = Data;
        ++Top;
    }
    return ( DataTable [ Top-1 ] );
}

```

```

template < class KEYTYPE, class DATATYPE>
INT FixArray<KEYTYPE, DATATYPE> :: IndexOf(
    KEYTYPE &Key)
{
    INT Index = (Top / 2);
    INT Shift = (Top / 2 - 1);

    if (!KeyTable)
        return ( -1 );

    if ( ! Top )
        return ( -1 );

    while ( KeyTable [ Index ] != Key )
    {
        if ( KeyTable [ Index ] > Key )
            Index -= Shift;
        else
            Index += Shift;

        if ( Index >= (INT)Top )

```

```

        Index = Top - 1; break; });
    if ( Index <= 0 )
        Index = 0; break; });
    if ( Shift == 0 )
        break; }
    Shift = Shift / 2;
}
if ( KeyTable [ Index ] == Key )
    return ( Index );

if ( KeyTable [ Index ] < Key )
{
    while ( Index < (INT)Top )
    {
        if ( KeyTable [ Index ] == Key )
            return ( Index );
        if ( KeyTable [ Index ] > Key )
            return ( -1 );
        ++Index;
    }
    return ( -1 );
}
else
{
    while ( Index >= 0 )
    {
        if ( KeyTable [ Index ] == Key )
            return ( Index );
        if ( KeyTable [ Index ] < Key )
            return ( -1 );
        --Index;
    }
    return ( -1 );
}
return ( -1 );
}

```

```

template <class KEYTYPE, class DATATYPE>
BOOL FixedArray<KEYTYPE, DATATYPE> :: RemoveAt(
    INT Index)
{
    if ( Index >= Top )
        return ( FALSE );

    if ( DataTable )
    {
        if ( (Index + 1) != Top )
            memcpy( (void*)&DataTable [ Index ],
                    (void*)&DataTable [ Index + 1 ],
                    sizeof ( DATATYPE ) * Top - Index - 1);
    }
    if ( KeyTable )
    {
        if ( (Index + 1) != Top )
            memcpy( (void*)&KeyTable [ Index ],
                    (void*)&KeyTable [ Index + 1 ],
                    sizeof ( KEYTYPE ) * Top - Index - 1);
    }
    --Top;
    return ( TRUE );
}

```

```

/*****
*
* Copyright (C) 2000, Louisiana State University, School of Medicine
*
* This C++ object library was developed from the University of California,
* Davis UCDMC DICOM Network Transport Libraries, in full compliance
* with the free access licence and copyright note below. The DICOM object
* library, however, contains conceptual deviations from the UCDMC library,
* as well as important bug and performance fixes, and CANNOT be
* used/modified/distributed/modified without our permission.
*
* Technical Contact: oleg@bit.csc.lsu.edu
*
*****/

```

```

/*****
*

```

```

* PQueue Unit Class (binary tree - based)
*
* ANSI (AFN) C++ Compatible / Templates Required
*

```

```

* usage:
*

```

```

* #include "pqueue.h"
*
* PQueue<ClassType> VarName;
*

```

```

* notes:
*

```

```

* Any Class used as the datatype must support the operators < > =
*****/

```

```

template <class DATATYPE> class PQueue : public Array<DATATYPE>
{
    DATATYPE dt;
public:
    DATATYPE & Push(DATATYPE &);
    DATATYPE & Pop();
};

```

```

template <class DATATYPE> class PQueueOfPtr : public Array<DATATYPE>
{
    DATATYPE dt;
public:
    DATATYPE & Push(DATATYPE &);
    DATATYPE & Pop();
};

```

```

/*****
*
* Template P-Queue Class implementation
*
*****/

```

```

template <class DATATYPE>
DATATYPE & PQueue<DATATYPE> :: Push(DATATYPE &Value)
{
    unsigned int Index;
    //unsigned int Base;
    DATATYPE tdt;

    Array<DATATYPE> :: Add(Value);

    Index = Array<DATATYPE> :: GetSize();
    --Index;
    while (Index)
    {
        if (Array<DATATYPE> :: Get(Index) < Array<DATATYPE> :: Get((Index-1)>>1))
        {
            tdt = Array<DATATYPE> :: Get(Index);
            Array<DATATYPE> :: Get(Index) = Array<DATATYPE> :: Get((Index-1)>>1);
            Array<DATATYPE> :: Get((Index-1)>>1) = tdt;
            Index = (Index-1) >> 1;
        }
        else
            break;
    }
}

```

```

return Value );
}

template <class DATATYPE>
DATATYPE & PQueue<DATATYPE> :: Pop()
{
    DATATYPE tdt;
    //DATATYPE tdt2;
    //unsigned int Index;
    unsigned int sorted;
    unsigned int Pick;
    unsigned int Child1;
    unsigned int Child2;
    unsigned int Hole;

    if(!Array<DATATYPE>::GetSize())
        return ( dt ); // error

    dt = Array<DATATYPE> :: Get ( 0 );

    if(Array<DATATYPE> :: GetSize() == 1)
    {
        Array<DATATYPE> :: RemoveAt ( 0 );
        return ( dt );
    }

    tdt = Array<DATATYPE> :: Get ( Array<DATATYPE> :: GetSize() - 1);
    Array<DATATYPE> :: RemoveAt ( Array<DATATYPE> :: GetSize() - 1);
    Hole = 0;
    sorted = 0;
    while(!sorted)
    {
        Child1 = 2 * Hole + 1;
        Child2 = 2 * Hole + 2;
        if(Child2 >= Array<DATATYPE> :: GetSize())
        {
            if ( Child1 >= Array<DATATYPE> :: GetSize())
            {
                Array<DATATYPE> :: Get(Hole) = tdt;
                return ( dt );
            }
            if(Array<DATATYPE> :: Get ( Child1 ) < tdt)
            {
                Array<DATATYPE> :: Get(Hole) = Array<DATATYPE> :: Get(Child1);
                Array<DATATYPE> :: Get(Child1) = tdt;
                return(dt);
            }
            Array<DATATYPE> :: Get(Hole) = tdt;
            return ( dt );
        }
        Pick = Child1;
        if ( Array<DATATYPE> :: Get ( Child1 ) > Array<DATATYPE> :: Get(Child2))
            Pick = Child2;
        if ( Array<DATATYPE> :: Get ( Pick ) < tdt)
        {
            Array<DATATYPE> :: Get ( Hole) = Array<DATATYPE> :: Get( Pick );
            Hole = Pick;
        }
        else
        {
            Array<DATATYPE> :: Get(Hole) = tdt;
            return ( dt );
        }
    }
    return ( dt );
}

template <class DATATYPE>
DATATYPE & PQueueOfPtr<DATATYPE> :: Push(DATATYPE &Value)
{
    unsigned int Index;
    //unsigned int Base;
    DATATYPE tdt;

    Array<DATATYPE> :: Add(Value);

    Index = Array<DATATYPE>::GetSize();
    --Index;
    while(Index)
    {
        if(*Array<DATATYPE>::Get(Index)) < (*Array<DATATYPE>::Get((Index-1)>>1))
    }

```

```

tdt = Array<DATATYPE> :: Get(Index);
Array<DATATYPE> :: Get(Index) = Array<DATATYPE> :: Get((Index-1)>>1);
Array<DATATYPE> :: Get((Index-1)>>1) = tdt;
Index = (Index-1) >> 1;

```

```

break;

```

```

return Value );

```

```

template class DATATYPE>
DATATYPE QueueOfPtr<DATATYPE> :: Pop()

```

```

DATATYPE tdt;
//DATATYPE tdt2;
//unsigned int Index;
unsigned int sorted;
unsigned int Pick;
unsigned int Child1;
unsigned int Child2;
unsigned int Hole;

```

```

if(!Array<DATATYPE> :: GetSize())
return ( dt ); // error

```

```

dt = Array<DATATYPE> :: Get ( 0 );

```

```

if(Array<DATATYPE> :: GetSize() == 1)

```

```

Array<DATATYPE> :: RemoveAt ( 0 );
return ( dt );

```

```

tdt = Array<DATATYPE> :: Get ( Array<DATATYPE> :: GetSize() - 1);
Array<DATATYPE> :: RemoveAt ( Array<DATATYPE> :: GetSize() - 1);

```

```

Hole = 0;

```

```

sorted = 0;

```

```

while (sorted)

```

```

{
Child1 = 2 * Hole + 1;

```

```

Child2 = 2 * Hole + 2;

```

```

if(Child2 >= Array<DATATYPE> :: GetSize())

```

```

{
if ( Child1 >= Array<DATATYPE> :: GetSize())

```

```

{
Array<DATATYPE> :: Get(Hole) = tdt;

```

```

return ( dt );
}

```

```

if((Array<DATATYPE> :: Get ( Child1 )) < (*tdt))

```

```

{
Array<DATATYPE> :: Get(Hole) = Array<DATATYPE> :: Get(Child1);

```

```

Array<DATATYPE> :: Get(Child1) = tdt;

```

```

return(dt);
}

```

```

Array<DATATYPE> :: Get(Hole) = tdt;

```

```

return ( dt );
}

```

```

Pick = Child1;

```

```

if ( (*Array<DATATYPE> :: Get ( Child1 )) > (*Array<DATATYPE> :: Get(Child2)))

```

```

Pick = Child2;

```

```

if ( (*Array<DATATYPE> :: Get ( Pick )) < (*tdt))

```

```

{
Array<DATATYPE> :: Get ( Hole) = Array<DATATYPE> :: Get( Pick );

```

```

Hole = Pick;
}

```

```

Array<DATATYPE> :: Get(Hole) = tdt;

```

```

return ( dt );
}

```

```

return ( dt );

```

```

// serviceclass.h: interface for the ServiceClass class.
//
////////////////////////////////////////////////////////////////////
#ifndef SERVICE_CLASS_H_INCLUDED_
#define SERVICE_CLASS_H_INCLUDED_

#include "Dicomview.h"

class ServiceClass
{
public:
    const static BYTE    PatientRoot;
    const static BYTE    StudyRoot;
    const static BYTE    PatientStudyRoot;
    const static BYTE    NormalPriority;
    const static BYTE    HighPriority;
    const static BYTE    LowPriority;

    const static UINT16  CEchoReq;
    const static UINT16  CEchoRsp;
    const static UINT16  CFindReq;
    const static UINT16  CFindRsp;
    const static UINT16  CGetReq;
    const static UINT16  CGetRsp;
    const static UINT16  CMoveReq;
    const static UINT16  CMoveRsp;
    const static UINT16  CStoreReq;
    const static UINT16  CStoreRsp;
    const static UINT16  CCancelReq;
    const static UINT16  CUnknownReq;

    const static UINT16  DataPresent;
    const static UINT16  DataAbsent;

    const static UINT16  StatusSuccess;
    const static UINT16  StatusPending;
    const static UINT16  StatusPendingOptionalKeys;
    const static UINT16  StatusCancelled;
    const static UINT16  StatusOutOfResources;
    const static UINT16  StatusCannotMatch;
    const static UINT16  StatusCannotPerformSuboperation;
    const static UINT16  StatusDestinationUnknown;
    const static UINT16  StatusDoesNotMatchSOP;
    const static UINT16  StatusFailedSubOperation;
    const static UINT16  StatusElementsDiscarded;
    const static UINT16  StatusDataDoesNotMatchSOP;
    const static UINT16  StatusUnableToProcess;

    ServiceClass(DICOMViewLog& dvl, DICOMDatabase& dtb)
        : m_Log(dvl), m_DBase(dtb) {} // parameters are passed by reference !
    ServiceClass(DICOMViewLog& dvl, DICOMDatabase& dtb, CallBackObject& cbo)
        : m_Log(dvl), m_DBase(dtb), m_ServiceCallback(cbo) {}
    virtual ~ServiceClass() {}

protected:
    DICOMViewLog&    m_Log;
    DICOMDatabase&   m_DBase;
    CallBackObject   m_ServiceCallback;

    bool    Receive(PDU_Service &PDU, DICOMCommandObject *DCO,
                   DICOMDataObject *DDO, const char* service);
    bool    Send(PDU_Service &PDU, DICOMCommandObject *DCO,
                DICOMDataObject *DDO, const char* service);
    bool    CheckStatus(DICOMCommandObject *DCO);
};

class BQUip : public ServiceClass
{
public:
    bool    CEcho(UINT ae_index);
    bool    CFind( UINT ae_index, DICOMDataObject& ddo_mask,
                  Array<DICOMDataObject> & ddo_found,
                  BYTE root, UINT16 Priority=NormalPriority);
    bool    CGet(UINT ae_index, DICOMDataObject& ddo_mask,
                  BYTE root, UINT16 Priority=NormalPriority);
    bool    CMove(UINT ae_index, DICOMDataObject& ddo_mask, char* destAET,
                  BYTE root, UINT16 Priority=NormalPriority);
    bool    CStore(PDU_Service &PDU, DICOMDataObject& DDO,

```

```

        UINT16 Priority=NormalPriority, char* originatorAETitle=NULL,
        UINT16 originatorPageID=0x0000);
    DICOMViewLog& dvl, DICOMDatabase& dtb);
    DICOMViewLog& dvl, DICOMDatabase& dtb, CallBackObject& cbo);
    ~SCPUser();

private:
    bool CancelIfNeeded(PDU_Service& PDU, UINT16 mID);
    bool GetAE(ApplicationEntity& ae, UINT ae_index);
    bool CCancel(PDU_Service& PDU, UINT16 mid);
    bool CEcho(ApplicationEntity& AE);
    bool CEcho(PDU_Service& PDU);
    bool CFind(ApplicationEntity &AE, DICOMDataObject& ddo_mask,
        Array<DICOMDataObject> & ddo_found,
        BYTE root, UINT16 Priority=NormalPriority);
    bool CFind(PDU_Service &PDU, DICOMDataObject& DDO,
        char* SOP, Array<DICOMDataObject> & found_list,
        UINT16 Priority=NormalPriority);
    bool CGet(ApplicationEntity &AE, DICOMDataObject& ddo_mask,
        BYTE root, UINT16 Priority=NormalPriority);
    bool CGet(PDU_Service &PDU, DICOMDataObject& DDO,
        char* SOP, UINT16 Priority=NormalPriority);
    bool CMove(ApplicationEntity &AE, DICOMDataObject& ddo_mask,
        char* destAET, BYTE root, UINT16 Priority=NormalPriority);
    bool CMove(PDU_Service &PDU, DICOMDataObject& DDO,
        char* SOP, char* destAET, UINT16 Priority=NormalPriority);
    bool VerifyCommandAndID(DICOMCommandObject& DCO,
        UINT16 Command, UINT16 MessageID);
    int ReceiveAndVerify(PDU_Service &PDU, DICOMCommandObject &DCO,
        DICOMDataObject &DDO, UINT16 Command,
        UINT16 MessageID, const char* service);
};

class SCPProvider : public ServiceClass
{
public:
    bool Echo(PDU_Service& PDU, DICOMCommandObject& DCO);
    bool CFind(PDU_Service &PDU, DICOMCommandObject &DCO, UINT16* cancelledID);
    bool CGet(PDU_Service &PDU, DICOMCommandObject &DCO, UINT16* cancelledID);
    bool CMove(PDU_Service &PDU, DICOMCommandObject &DCO, UINT16* cancelledID);
    bool CStore(PDU_Service &PDU, DICOMCommandObject &DCO);
    bool IdentifyAndProcess(PDU_Service &PDU, DICOMCommandObject& DCO, UINT16* cancelledID);

    SCPProvider(DICOMViewLog& dvl, DICOMDatabase& dtb);
    SCPProvider(DICOMViewLog& dvl, DICOMDatabase& dtb, CallBackObject& cbo);
    virtual ~SCPProvider();

private:
    bool ContainsData(DICOMCommandObject& DCO);
};

#endif // defined(_SERVICE_CLASS_H_INCLUDED_)

```

```

/*****
 * Copyright (C) 2000, Louisiana State University, School of Medicine
 *
 * This DICOM object library was developed from the University of California,
 * San Diego UCDMC DICOM Network Transport Libraries, in full compliance
 * with the free access licence and copyright note below. The DICOM object
 * library however contains conceptual deviations from the UCDMC library,
 * as well as important bug and performance fixes, and CANNOT be
 * used, copied, distributed/modified without our permission.
 *
 * Technical Contact: oleg@bit.csc.lsu.edu
 *
 *****/
#ifdef _UTIL_H_INCLUDED_
#  ifndef _UTIL_H_INCLUDED_

/* Utility functions */
UINT      uniq();
UINT32    uniq32();
UINT16    uniq16();
UINT8     uniq8();
UINT8     uniq8odd();
UINT16    uniq16odd();

inline void ZeroMem(BYTE* mem, UINT Count) { memset((void *) mem, 0, Count); };
inline void ZeroMem(char* mem, UINT Count) { memset((void *) mem, 0, Count); };
void Flip2DBufferY(BYTE* buf, UINT32 buf_size,
                  UINT32 nrows);
bool CreateAndSetCurrentDirectory(char *dir);
bool RemoveDICOMSubstring(char* str, char*sub);
bool AddDICOMSubstring(char* str, char*sub);
bool SerializeString(FILE* fp, char* str, bool is_loading);
bool Serializebool(FILE* fp, bool& x, bool is_loading);
bool SerializeBOOL(FILE* fp, BOOL& x, bool is_loading);
bool SerializeBYTE(FILE* fp, BYTE& x, bool is_loading);
bool SerializeInteger(FILE* fp, int& x, bool is_loading);
bool SerializeUINT(FILE* fp, UINT& x, bool is_loading);
bool SerializeDouble (FILE* fp, double& x, bool is_loading);
inline bool IsEmptyString(char* s)
{
    if(s==NULL) return true;
    if(s[0]==0) return true;
    return false;
};
inline bool IsUniqueString(char* s)
{
    if(IsEmptyString(s)) return false;
    if(strchr(s,'*') || strchr(s,'?') ||
       strchr(s,'\\')) return false;
    return true;
};
inline bool IsBlankString(char* s)
{
    if(IsEmptyString(s)) return true;
    for(UINT i=0; i<strlen(s); i++)
    {
        if(s[i]!=' ') return false;
    }
    return true;
};

BOOL      ByteCopy(BYTE *, BYTE *, UINT);
char*     GetShortFileName(char* fullpathname);
char*     GetDICOMSubstring(BYTE* str, UINT32 str_len,
                           char* sub, UINT32 sub_len, int number=1);
char*     GetDICOMSubstring(char* str, char* sub, UINT32 sub_len,
                           int number=1);
int       FindChar(char* str, char c);
int       FindCharReverse(char* str, char c);
UINT      ByteStrLength(BYTE *);
long      FileLength(char* filename);

#endif

```





```

/*****
 * Copyright (C) 2000, Louisiana State University, School of Medicine
 *
 * This DICOM object library was developed based on University of California,
 * San Diego UCDMC DICOM Network Transport Libraries, in full compliance
 * with the copyright note below. This version however contains conceptual
 * changes from the UCDMC library, as well as important bug and performance
 * fixes and cannot be used/copied/distributed without our permission
 *
 * Technical Contact: oleg@bit.csc.lsu.edu
 *
 *****/
# include "dicom.hpp"

/*****
 *
 * Application Context Class
 *
 *****/
ApplicationContext :: ApplicationContext()
{
    ItemType = 0x10;
    Reserved1 = 0;
    Length=0;
}

ApplicationContext :: ApplicationContext(UID &uid)
{
    ItemType = 0x10;
    Reserved1 = 0;
    Length=0;
    ApplicationContextName = uid;
}

ApplicationContext :: ApplicationContext(BYTE *name)
{
    ItemType = 0x10;
    Reserved1 = 0;
    Length=0;
    ApplicationContextName.Set(name);
}

ApplicationContext :: ~ApplicationContext()
{
    // nothing to de-allocate specifically
}

void ApplicationContext :: Set(UID &uid)
{
    ApplicationContextName = uid;
}

void ApplicationContext :: Set(BYTE *name)
{
    ApplicationContextName.Set(name);
}

BOOL ApplicationContext :: Write(Buffer &Link)
{
    if((Length) Size();
    Link << ItemType;
    Link << Reserved1;
    Link << Length;
    Link.Write((BYTE *) ApplicationContextName.GetBuffer(), Length);
    Link.Flush();
    return ( TRUE );
}

BOOL ApplicationContext :: Read(Buffer &Link)
{
    Link >> ItemType;
    return ( this->ReadDynamic(Link) );
    return ( TRUE );
}

BOOL ApplicationContext :: ReadDynamic(Buffer &Link)
{
    Link >> Reserved1;
    Link >> Length;
    Link.Read((BYTE *) ApplicationContextName.GetBuffer(), Length);
    ApplicationContextName.GetBuffer()[Length] = '\0';
}

```

```

        AbstractSyntaxName.SetLength(Length);
    }

UINT AbstractSyntaxContext :: Size()
{
    return ApplicationContextName.GetSize();
    return sizeof(BYTE) + sizeof(BYTE) + sizeof(UINT16) + Length;
}

/*****
 * Abstract Syntax Class
 *****/
AbstractSyntax :: AbstractSyntax()
{
    ItemType = 0x30;
    Reserved1 = 0;
    Length = 0;
}

AbstractSyntax :: AbstractSyntax(UINT &uid)
{
    ItemType = 0x30;
    Reserved1 = 0;
    Length = 0;
    AbstractSyntaxName = uid;
}

AbstractSyntax :: AbstractSyntax(BYTE *name)
{
    ItemType = 0x30;
    Reserved1 = 0;
    Length = 0;
    AbstractSyntaxName.Set(name);
}

AbstractSyntax :: ~AbstractSyntax()
{
    // nothing to de-allocate specifically
}

void AbstractSyntax :: Set(UINT &uid)
{
    AbstractSyntaxName = uid;
}

void AbstractSyntax :: Set(BYTE *name)
{
    AbstractSyntaxName.Set(name);
}

BOOL AbstractSyntax :: Write(Buffer &Link)
{
    return (Length) Size();
    Link.ItemType;
    Link.Reserved1;
    Link.Length;
    Link.Write((BYTE *) AbstractSyntaxName.GetBuffer(), Length);
    Link.Flush();
    return (TRUE);
}

BOOL AbstractSyntax :: Read(Buffer &Link)
{
    Link.ItemType;
    return (this->ReadDynamic(Link));
}

BOOL AbstractSyntax :: ReadDynamic(Buffer &Link)
{
    Link.Reserved1;
    Link.Length;
    Link.Read((BYTE *) AbstractSyntaxName.GetBuffer(), Length);
    AbstractSyntaxName.GetBuffer()[Length] = '\0';
    AbstractSyntaxName.SetLength(Length);
    return (TRUE);
}

UINT AbstractSyntax :: Size()
{
    return AbstractSyntaxName.GetSize();
}

```

```

    sizeof(BYTE) + sizeof(BYTE) + sizeof(UINT16) + Length
)

/*****
 * Transfer Syntax
 *****/
TransferSyntax :: TransferSyntax()
{
    ItemType = 0x40;
    Reserved1 = 0;
    Length = 0;
}

TransferSyntax :: TransferSyntax(UID &uid)
{
    ItemType = 0x40;
    Reserved1 = 0;
    Length = 0;
    TransferSyntaxName = uid;
}

TransferSyntax :: TransferSyntax(BYTE *name)
{
    ItemType = 0x40;
    Reserved1 = 0;
    Length = 0;
    TransferSyntaxName.Set(name);
}

TransferSyntax :: ~TransferSyntax()
{
    // Nothing to de-allocate specifically
};

void TransferSyntax :: Set(UID &uid)
{
    TransferSyntaxName = uid;
}

void TransferSyntax :: Set(BYTE *name)
{
    TransferSyntaxName.Set(name);
}

bool TransferSyntax :: Write(Buffer &Link)
{
    Link.Length().Size();
    Link.ItemType();
    Link.Reserved1();
    Link.Length();
    Link.Write((BYTE *) TransferSyntaxName.GetBuffer(), Length);
    Link.Flush();
    return ( TRUE );
}

bool TransferSyntax :: Read(Buffer &Link)
{
    Link.ItemType();
    return ( this->ReadDynamic(Link) );
}

bool TransferSyntax :: ReadDynamic(Buffer &Link)
{
    Link.ItemType();
    Link.Length();
    Link.Read((BYTE *) TransferSyntaxName.GetBuffer(), Length);
    TransferSyntaxName.GetBuffer()[Length] = '\0';
    TransferSyntaxName.SetLength(Length);
    return ( TRUE );
}

UINT16 TransferSyntax :: Size()
{
    return ( TransferSyntaxName.GetSize() );
}

return ( sizeof(BYTE) + sizeof(BYTE) + sizeof(UINT16) + Length );
}

/*****
 *
 *****/

```

```

* Implementation Class
*
*****/
ImplementationClass :: ImplementationClass()
{
    m_ItemType = 0x52;
    m_Reserved1 = 0;
    m_Length = 0;
    m_ImplementationName.Set((BYTE*) IMPLEMENTATION_CLASS_STRING);
}
ImplementationClass :: ImplementationClass(UID &uid)
{
    m_ItemType = 0x52;
    m_Reserved1 = 0;
    m_Length = 0;
    m_ImplementationName = uid;
}
ImplementationClass :: ImplementationClass(BYTE *name)
{
    m_ItemType = 0x52;
    m_Reserved1 = 0;
    m_Length = 0;
    m_ImplementationName.Set(name);
}
ImplementationClass :: ~ImplementationClass()
{
    // Nothing to de-allocate specifically
};

void ImplementationClass :: Set(UID &uid)
{
    m_ImplementationName = uid;
}
void ImplementationClass :: Set(BYTE *name)
{
    m_ImplementationName.Set(name);
}
BOOL ImplementationClass :: Write(Buffer &Link)
{
    m_Length = Size();
    m_ItemType = ItemType;
    m_Reserved1 = Reserved1;
    m_Length = Length;
    Link.Write((BYTE *) ImplementationName.GetBuffer(), Length);
    Link.Flush();
    return (TRUE);
}
BOOL ImplementationClass :: Read(Buffer &Link)
{
    Link = ItemType;
    return (this->ReadDynamic(Link));
}
BOOL ImplementationClass :: ReadDynamic(Buffer &Link)
{
    Link = Reserved1;
    m_Length = Length;
    Link.Read((BYTE *) ImplementationName.GetBuffer(), Length);
    ImplementationName.GetBuffer()[Length] = '\0';
    m_ImplementationName.SetLength(Length);
    return (TRUE);
}
UINT ImplementationClass :: Size()
{
    m_Length = ImplementationName.GetSize();
    return (sizeof(BYTE) + sizeof(BYTE) + sizeof(UINT16) + Length);
}

/*****
*
* Implementation Class
*
*****/
ImplementationVersion :: ImplementationVersion()
{

```

```

    itemType = 0x55;
    reserved1 = 0;
    return (BYTE*) IMPLEMENTATION_VERSION_STRING;
}

ImplementationVersion :: ImplementationVersion(UID &uid)
{
    itemType = 0x55;
    reserved1 = 0;
    return uid;
}

ImplementationVersion :: ImplementationVersion(BYTE *name)
{
    itemType = 0x55;
    reserved1 = 0;
    return name;
}

ImplementationVersion :: ~ImplementationVersion()
{
    // to de-allocate specifically
};

void ImplementationVersion :: Set(UID &uid)
{
    return uid;
};

void ImplementationVersion :: Set(BYTE *name)
{
    return name;
};

bool ImplementationVersion :: Write(Buffer &Link)
{
    itemType = Size();
    reserved1 = 0;
    Length = Length;
    Link.Write((BYTE *) Version.GetBuffer(), Length);
    Link.Flush();
    return (TRUE);
}

bool ImplementationVersion :: Read(Buffer &Link)
{
    itemType = 0;
    return (this->ReadDynamic(Link));
}

bool ImplementationVersion :: ReadDynamic(Buffer &Link)
{
    reserved1 = 0;
    Length = Length;
    Link.Read((BYTE *) Version.GetBuffer(), Length);
    Version.GetBuffer()[Length] = '\0';
    Version.SetLength(Length);
    return (TRUE);
}

UINT16 ImplementationVersion :: Size()
{
    return Version.GetSize();
}

/*****
 * Presentation Context
 *****/
PresentationContext :: PresentationContext()
{
    itemType = 0x20;
    reserved1 = 0;
    PresentationContextID = uniq8odd();
    reserved2 = 0;
    reserved3 = 0;
    reserved4 = 0;
}

```

```

}
PresentationContext :: PresentationContext( AbstractSyntax &Abs,
                                           TransferSyntax &Tran)
{
    m_Abs = Abs;
    m_TransId (Tran);
    m_AbsId = x20;
    m_AbsId = 0;
    m_PresentationContextID = uniq8odd();
    m_Reserved1 = 0;
    m_Reserved2 = 0;
    m_Reserved3 = 0;
    m_Reserved4 = 0;
}

PresentationContext :: ~PresentationContext()
{
}

void PresentationContext :: SetAbstractSyntax( AbstractSyntax &Abs)
{
    m_Abs = Abs;
}

void PresentationContext :: AddTransferSyntax( TransferSyntax &Tran)
{
    m_Trans.Add ( Tran );
}

bool PresentationContext :: Write ( Buffer &Link )
{
    m_AbsId = Size();
    m_AbsId = ItemType;
    m_AbsId = Reserved1;
    m_AbsId = Length;
    m_AbsId = PresentationContextID;
    m_AbsId = Reserved2;
    m_AbsId = Reserved3;
    m_AbsId = Reserved4;
    m_AbsId.Write(Link);
    Link.Flush();
    m_AbsId = 0;
    while ( Index < TrnSyntax.GetSize() )
    {
        m_AbsId[Index].Write(Link);
        Index++;
    }
    if ( Index ) return TRUE;
    return FALSE;
}

bool PresentationContext :: Read ( Buffer &Link)
{
    m_AbsId = ItemType;
    return ( this->ReadDynamic(Link) );
}

bool PresentationContext :: ReadDynamic (Buffer &Link)
{
    m_AbsId = Count;
    m_AbsId = TransferSyntax Tran;

    m_AbsId = Reserved1;
    m_AbsId = Length;
    m_AbsId = PresentationContextID;
    m_AbsId = Reserved2;
    m_AbsId = Reserved3;
    m_AbsId = Reserved4;

    m_AbsId = Length - sizeof(BYTE) - sizeof(BYTE) - sizeof(BYTE) - sizeof(BYTE);
    m_AbsId.Read(Link);
    m_AbsId = Count - AbsSyntax.Size();
    while ( Count > 0 )
    {
        Tran.Read ( Link );
        m_AbsId = Count - Tran.Size();
        m_AbsId.Add ( Tran );
    }
    return TRUE;
    return FALSE;
}

```

```

)
UINT32 CTrnSyntaxContext::Size()
{
    return sizeof(BYTE) + sizeof(BYTE) + sizeof(BYTE) + sizeof(BYTE);
    TrnSyntax.Size();
    Length = 0;
    TrnSyntax.GetSize()
    TrnSyntax.Get(Index).Size();
    Length;

    Length + sizeof(BYTE) + sizeof(BYTE) + sizeof(UINT16));
}

/*****
*
* Length
*
*****/

MaximumSubLength()
{
    Length = 0x51;
    Length = 0;
    Length = sizeof(UINT32);
    Length = 16384;
}

MaximumSubLength(UINT32 Max)
{
    Length = 0x51;
    Length = 0;
    Length = sizeof(UINT32);
    Length = Max;
}

MaximumSubLength() :: ~MaximumSubLength()
{
    // to de-allocate
}

void CTrnSyntaxContext::Set(UINT32 Max)
{
    Length = Max;
}

UINT32 CTrnSyntaxContext::Get()
{
    return Length;
}

bool CTrnSyntaxContext::Write(Buffer &Link)
{
    WriteType;
    WriteReserved1;
    WriteLength;
    WriteMaximumLength;
    Write();
    return TRUE;
}

bool CTrnSyntaxContext::Read(Buffer &Link)
{
    ReadType;
    ReadDynamic(Link);
}

bool CTrnSyntaxContext::ReadDynamic(Buffer &Link)
{
    ReadReserved1;
    ReadLength;
    ReadMaximumLength;
    Read();
}

UINT32 CTrnSyntaxContext::Size()
{
    return sizeof(UINT32);
    Length + sizeof(BYTE) + sizeof(BYTE) + sizeof(UINT16));
}

```



```

/*****
*
* SCPSCURoleSelect
*
*****/
SCPSCURoleSelect :: SCPSCURoleSelect()
{
    ItemLength = 0x54;
    Reserved1 = 0;
    ItemLength = 1;
}

SCPSCURoleSelect :: ~SCPSCURoleSelect()
{
    // Nothing to de-allocate
}

BOOL SCPSCURoleSelect :: Write(Buffer &Link)
{
    ItemLength = Size();
    ItemType = SCPSCURoleSelect;
    Reserved1 = 0;
    ItemLength = 1;
    TL = SOPuid.GetSize();
    Write((BYTE *) SOPuid.GetBuffer(), TL);
    Write((BYTE *) SCPSCURoleSelect, 1);
    Write((BYTE *) TRUE);
}

SCPSCURoleSelect :: Read(Buffer &Link)
{
    ItemType = SCPSCURoleSelect;
    return (this->ReadDynamic(Link));
}

SCPSCURoleSelect :: ReadDynamic(Buffer &Link)
{
    TL = Link.GetLength();
    Reserved1 = 0;
    ItemLength = 1;
    Read((BYTE *) SOPuid.GetBuffer(), TL);
    Read((BYTE *) SCPSCURoleSelect, 1);
    Read((BYTE *) TRUE);
}

UINT16 SCPSCURoleSelect :: Size()
{
    Length = sizeof(UINT16) + SOPuid.GetSize() + sizeof(BYTE) + sizeof(BYTE);
    return (Length + sizeof(BYTE) + sizeof(BYTE) + sizeof(UINT16));
}

/*****
*
* ExtendedNegotiation
*
*****/
ExtendedNegotiation :: ExtendedNegotiation()
{
    ItemLength = 0x56;
    Reserved1 = 0;
    OptionalDB = 1; // request relational DB support, by default
    CFind = 1; // C-Find, Patient Root, by default
}

ExtendedNegotiation :: ~ExtendedNegotiation()
{
    // Nothing to de-allocate
}

```

```

BOC1 ExtendedNegotiation :: Write(Buffer &Link)
{
    ItemSize = Size();
    ItemType = ItemType;
    Reserved1 = Reserved1;
    Length = Length;
    TL = SOPuid.GetSize();
    (char *) SOPuid.GetBuffer(), TL);
    RelationalDB;
    Flush();
    return (TRUE);
}

BOC1 ExtendedNegotiation :: Read(Buffer &Link)
{
    ItemType;
    this->ReadDynamic(Link);
}

BOC1 ExtendedNegotiation :: ReadDynamic(Buffer &Link)
{
    TL;
    Reserved1;
    Length;
    TL;
    (char *) SOPuid.GetBuffer(), TL);
    RelationalDB;
    Flush();
    return (TRUE);
}

UINT16 ExtendedNegotiation :: Size()
{
    return (sizeof(UINT16) + SOPuid.GetSize() + sizeof(BYTE) +
        Length + sizeof(BYTE) + sizeof(BYTE) + sizeof(UINT16));
}

*****
* User Information
*****/
UserInformation :: UserInformation()
{
    ItemType = 0x50;
    Reserved1 = 0;
    Language = 0;
    Flush();
}

UserInformation :: ~UserInformation()
{
    // Nothing to de-allocate
}

void UserInformation :: SetMax(MaximumSubLength &Max)
{
    MaxSubLength = Max;
}

UINT16 UserInformation :: GetMax()
{
    return (MaxSubLength.Get());
}

BOC1 UserInformation :: Write(Buffer &Link)
{
    ItemType;
    Reserved1;
    Length;
    Flush();
    MaxSubLength.Write(Link);
    Language.Write(Link);
    RelationalDB.Write(Link);
    Negotiation.Write(Link); - optional, needs to be initialized
    TRUE.Write(Link); - optional, needs to be initialized
    return (TRUE);
}

```

```

BOC: Information :: ReadDynamic(Buffer &Link)
{
    ...Type;
    ...->ReadDynamic(Link) );
}

```

```

BOC: Information :: ReadDynamic(Buffer &Link)

```

```

    TempByte;
    ... = 0;
    ...Reserved1;
    ...Length;
    ... = Length;
    ... = 0)

    TempByte;
    ... ( TempByte )

    ... 0x51: // Reading Max Sub Length
    MaxSubLength.ReadDynamic(Link);
    Count = Count - MaxSubLength.Size();
    break;
    ... 0x52: // Reading Implementation Class
    ImpClass.ReadDynamic(Link);
    Count = Count - ImpClass.Size();
    break;
    ... 0x54: // Role selection
    SCPSCURole.ReadDynamic(Link);
    Count = Count - SCPSCURole.Size();
    UserInfoBaggage += SCPSCURole.Size();
    break;
    ... 0x55: // Reading Implementation Version
    ImpVersion.ReadDynamic(Link);
    Count = Count - ImpVersion.Size();
    break;
    ... 0x56: // Reading Extended Negotiation
    ExtNegotiation.ReadDynamic(Link);
    Count = Count - ExtNegotiation.Size();
    break;
    ... // Unknown Packet
    Link.Kill(Count-1);
    UserInfoBaggage = Count;
    ... -1;

    ... return TRUE;
    ... FALSE;

```

```

BOC: Information :: Size()

```

```

    Count = MaxSubLength.Size();
    Count += ImpClass.Size();
    Count += ImpVersion.Size();
    Count += ExtNegotiation.Size();
    Count += SCPSCURole.Size();
    Count = Length + UserInfoBaggage + sizeof(BYTE) + sizeof(BYTE) + sizeof(UINT16));
}

```

```

/*****
*
* ... Packet
*
*****/

```

```

AAS: AssociateRQ :: AAssociateRQ()

```

```

{
    ... = 0x01;
    ... = 0;
    ...
    ...Version = 0x0001;
    ... = 0;
    ...CallingApTitle, 17);
    ...CallingApTitle, 17);
    ...Reserved3, 32);
}

```

```

AAS: AssociateRQ :: AAssociateRQ(BYTE *CallingAp, BYTE *CalledAp)

```

```

{
    ... = 0x01;

```

```

        Version = 0x0001;
        SetCalledApTitle(17);
        SetCallingApTitle(17);
        Reserved3(32);
        SetCallingApTitle(CallingAp, ByteStrLength(CallingAp));
        SetCalledApTitle(CalledAp, ByteStrLength(CalledAp));
    }
AAssociateRQ :: -AAssociateRQ()
{
    PresContexts.GetSize()
    PresContexts.Get ( 0 ).TrnSyntax.ClearType = TRUE;
    PresContexts.RemoveAt ( 0 );
    PresContexts.ClearType = TRUE;
}

void AAssociateRQ :: SetCalledApTitle(BYTE *CalledAp)
{
    SetCalledApTitle(17);
    SetCalledApTitle(CalledAp, ByteStrLength(CalledAp));
}

void AAssociateRQ :: SetCallingApTitle(BYTE *CallingAp)
{
    SetCallingApTitle(17);
    SetCallingApTitle(CallingAp, ByteStrLength(CallingAp));
}

void AAssociateRQ :: SetApplicationContext(ApplicationContext &AppC)
{
    SetContext AppC;
}

void AAssociateRQ :: SetApplicationContext(UID &uid)
{
    SetContext uid;
}

void AAssociateRQ :: AddPresentationContext(PresentationContext &PresContext)
{
    PresContexts.Add(PresContext);
    PresContexts.Get(PresContexts.GetSize()-1).TrnSyntax.ClearType = TRUE;
}

void AAssociateRQ :: SetUserInformation(UserInformation &User)
{
    SetContext User;
}

void AAssociateRQ :: ClearPresentationContexts()
{
    PresContexts.GetSize() PresContexts.RemoveAt ( 0 );
}

BOOFAAssociateRQ :: Write(Buffer &Link)
{
    Write(
        ItemType;
        Reserved1;
        Length;
        ProtocolVersion;
        Reserved2;
        SetContext(BYTE *) CalledApTitle, 16);
        SetContext(BYTE *) CallingApTitle, 16);
        SetContext(BYTE *) Reserved3, 32);
        Write(Link);
        PresContexts.GetSize()
        PresContexts[Index].Write(Link);
        Index;
        Write(Link);
        TRUE );
}

BOOFAAssociateRQ :: Read(Buffer &Link)

```

```

{
    Type;
    ReadDynamic(Link);
}

BOOL ReadWaterQ :: ReadDynamic(Buffer &Link)
{
    Count;
    TempByte;
    PresentationContext *PresContext;

    Reserved1;
    Length;
    ProtocolVersion;
    Reserved2;
    ReadDynamic((BYTE *) CalledApTitle, 16);
    ReadDynamic((BYTE *) CallingApTitle, 16);
    ReadDynamic((BYTE *) Reserved3, 32);
    CalledApTitle[16] = '\0';
    CallingApTitle[16] = '\0';

    Length = sizeof(UINT16) + sizeof(UINT16) + 16 + 16 + 32;
    if (Count > 0)
    {
        ReadDynamic((BYTE *) &TempByte, sizeof(BYTE));
        return TempByte;

        case 0x50: // User information
            UserInfo.ReadDynamic(Link);
            Count = Count - UserInfo.Size() - UserInfo.UserInfoBaggage;
            break;

        case 0x20: // Presentation context
            PresContext = new PresentationContext;
            PresContext->TrnSyntax.ClearType = TRUE;
            PresContext->ReadDynamic(Link);
            Count = Count - PresContext->Size();
            PresContexts.Add(*PresContext);
            PresContext->TrnSyntax.ClearType = FALSE;
            delete PresContext;
            break;

        case 0x10: // Application context
            AppContext.ReadDynamic(Link);
            Count = Count - AppContext.Size();
            break;

        default: // Unknown item
            Link.Kill(Count-1);
            Count = -1;
    }

    return TRUE;
}

FALSE;

UINT16 WaterQ :: Size()
{
    Length = sizeof(UINT16) + sizeof(UINT16) + 16 + 16 + 32;
    Length += AppContext.Size();

    Index = 0;
    while (Index < PresContexts.GetSize())
    {
        Length += PresContexts[Index].Size();
        Index++;
    }

    Length += UserInfo.Size();
    return Length + sizeof(BYTE) + sizeof(BYTE) + sizeof(UINT32);
}

```